Débogage symbolique multi-langages pour les plates-formes d'exécution généralistes

Damien Ciabrini

Université de Nice – Sophia Antipolis **UFR Sciences**
École Doctorale Sciences et Technologies de l'Information et de la Communication

THÈSE

Présentée pour obtenir le titre de

Docteur en Sciences
de l'Université de Nice – Sophia Antipolis

Spécialité Informatique

par

Damien CIABRINI

Débogage symbolique multi-langages pour les plates-formes d'exécution généralistes

Thèse dirigée par Manuel SERRANO
et préparée à l'INRIA Sophia, projet MIMOSA

Soutenue le 3 Octobre 2006 devant le jury composé de :

MM.	Michel RIVEILL	Président	UNSA
	Emmanuel CHAILLOUX	Rapporteur	PPS
	Christophe DONY	Rapporteur	LIRMM
	Christian QUEINNEC	Examinateur	LIP6
	Manuel SERRANO	Directeur de thèse	INRIA

Remerciements

Je remercie M. Michel Riveill pour avoir accepté de présider le jury de cette thèse. Je remercie aussi MM. Emmanuel Chailloux et Chritophe Dony pour avoir accepté la tâche de rapporteur de cette thèse, ainsi que M. Christian Queinnec qui a bien voulu en être l'examinateur.

Je remercie tout particulièrement M. Manuel Serrano, mon directeur de thèse, qui m'a beaucoup apporté durant ces quelques années passées à l'INRIA. Merci d'avoir toujours été à l'écoute lorsque j'en avais besoin et d'avoir su me guider tout en me laissant une grande liberté.

Je remercie M. Bernard Serpette pour son aide précieuse tout au long de mes travaux. Merci de m'avoir fait découvrir le code octet JVM, les subtilités de la compilation de Scheme vers la JVM, ainsi que les joies de la décompilation du code octet JavaCard vers Scheme !

Je remercie M. Erick Gallesio pour tout ce qu'il m'a appris sur l'histoire de l'informatique, sur ces veilles machines dont je ne soupçonnais même pas l'existence, ainsi que sur tous ces clones d'Emacs dont plus grand monde ne doit se souvenir.

Merci à tout le reste de l'équipe MIMOSA, qui m'a accueilli si chaleureusement pendant ma thèse. Merci à Gérard Boudol, notre chef de projet. Merci à Frédéric Boussinot pour avoir toujours pris le temps de lire mes manuscrits et de me les corriger. Et surtout, un grand merci à mes co-bureaux Christian Brunette, Stéphane Épardaud et Florian Loitsch pour tout ce que nous avous partagé ensemble.

Enfin, un Grand Merci à Mel qui m'a supporté durant la rédaction de cette thèse et qui a même eu le courage de la lire et d'en corriger des parties !

« *Le vrai courage, c'est celui de trois heures du matin.* »
Napoléon

Table des matières

Chapitre 1

Introduction

La conception de logiciel est une activité complexe qui se décline en une multitude de formes ; de la simple création d'utilitaires modestes à usage personnel, jusqu'à la création de très gros programmes conçus par plusieurs équipes de développeurs et distribués à grande échelle.

En quelques dizaines d'années, la façon d'écrire et d'exécuter des programmes a beaucoup évolué. Les techniques de compilation modernes ont entraîné une utilisation de plus en plus fréquente des machines virtuelles. De nos jours, les programmes font couramment usage de *scripts*, ces fragments de code source aisément modifiables et qui sont interprétés à l'exécution. Avec la généralisation des réseaux, certains programmes [Esp04] sont entièrement exécutés sur des serveurs distants.

Les langages de programmation se sont adaptés à ces nouveaux contextes d'utilisation. Ainsi, ces dernières années, de nombreux langages « niches » ont été conçus pour fournir de nouvelles abstractions de haut niveau capables de répondre aux besoins spécifiques de certains domaines d'application : réseaux, sécurité, concurrence, etc...

Avec la progression des techniques de compilation et la progression de la puissance des ordinateurs, les langages de haut niveau sont de plus en plus répandus. De plus, le besoin d'inter-opérabilité et la convergence des plates-formes d'exécution fait que les applications récentes combinent souvent plusieurs langages de programmation.

1.1 Présentation du débogage

Quelle que soit la méthodologie ou le langage de programmation utilisé durant la conception d'un logiciel, il existe une tâche à laquelle chaque programmeur est un jour confronté : corriger son programme parce qu'il ne fonctionne pas correctement.

En général, on se rend compte qu'un programme ne fonctionne pas correctement au moment où il présente des symptômes de dysfonctionnement. La cause de ces symptômes est en fait la présence de fautes dans le code source du programme ou la mauvaise spécification du traitement à opérer. Ces fautes sont appelées des *bogues*. Le débogage est l'action de traiter les causes du dysfonctionnement : lorsqu'une erreur se manifeste à l'exécution, on cherche à localiser le bogue qui en est responsable et à le corriger. Un débogueur est un programme qui fournit au programmeur un ensemble d'outils pour faciliter la compréhension des bogues et leur localisation dans le code source.

La phase de correction du programme est primordiale dans la mise au point de logiciel et elle est bien connue pour être difficile. Selon une phrase célèbre de Brian Kernighan,

« Le débogage est deux fois plus difficile que l'écriture de code. Donc, si l'on écrit du code aussi intelligemment que possible, on n'est, par définition, pas assez intelligent pour le déboguer. ». Plus sérieusement, de nombreuses études empiriques montrent que plus de la moitié du temps et des coûts de développement d'un programme peuvent être consacrés à sa mise au point [Zel78, Boe81, CvM93].

Il existe plusieurs critères qui permettent de considérer qu'un programme est incorrect. Citons ci-dessous quelques uns des critères les plus couramment rencontrés :

- l'exécution du programme provoque une erreur systématique lorsqu'il traite un ensemble de données particulières en entrée. C'est sans doute le type de bogue le plus simple à traiter car il est reproductible si l'on relance le programme avec les mêmes entrées ;
- l'exécution du programme ne provoque pas d'erreur, mais le résultat retourné est invalide, ou la précision de ce résultat n'est pas conforme aux attentes. C'est le signe que le programme est « algorithmiquement » faux ;
- le programme provoque un plantage de manière non déterministe pour une entrée particulière. C'est le signe que des événements extérieurs au programme (par exemple l'ordonnanceur du système d'exploitation) influent sur l'exécution du programme et peuvent l'amener à une configuration qui entraîne une erreur. Ce type de bogue est difficile à caractériser et à reproduire car les événements perturbateurs ne sont pas contrôlables ;
- la consommation mémoire durant l'exécution n'est pas conforme aux attentes. Ce type de problème apparaît en général lorsqu'une partie du programme ne libère pas toute la mémoire qu'elle avait allouée pour ses besoins et peut entraîner des plantages si la quantité de mémoire disponible devient insuffisante ;
- l'exécution des programmes est anormalement lente. Cela peut indiquer qu'une partie du programme répète plus souvent que nécessaire un calcul coûteux en temps. Bien que ces ralentissement n'entraînent pas de plantage, fixer leur cause s'avère souvent primordial pour le bon fonctionnement du programme.

À la vue des différents symptômes évoqués ci-dessus, il est clair que les types de bogues pouvant se manifester à l'exécution sont nombreux et souvent de nature très différente. Il existe donc différentes réponses dans le but d'obtenir des programmes corrects.

La première réponse repose sur une approche préventive, dans laquelle on cherche à s'assurer *avant* l'exécution qu'un programme est correct par rapport à certains critères. Cette approche est basée sur l'utilisation d'analyses statiques qui permettent de détecter automatiquement des erreurs en se basant sur des propriétés logiques d'un programme. Certaines analyses classiques sont directement intégrées au compilateur, comme par exemple la détection d'erreur de typage [PL99]. D'autres analyses sont disponibles sous formes d'outils externes [Joh79] et servent en général à détecter des erreurs subtiles ou à prouver, en l'absence d'erreur, qu'un programme fonctionne correctement. Les analyses statiques sont utiles pour détecter des erreurs logiques, mais elles ne sont pas capables de détecter qu'un programme est « algorithmiquement » faux. D'autres analyses sont tout simplement trop coûteuses en ressources pour être appliquées efficacement sur de très gros programmes.

Les analyses statiques ne permettent donc pas de remplacer le processus de débogage proprement dit, qui consiste à constater la présence d'un bogue à l'exécution, puis à en rechercher la cause pour le corriger. Il existe différentes approches de débogage. Toutes les approches instrumentent l'exécution originale du programme et chacune d'entre elles offre des avantages selon les contraintes d'utilisation du débogueur ou le type de bogue que l'on

souhaite corriger. On peut les regrouper en deux grandes catégories :

- *Le débogage par inspection.* Cette approche consiste à instrumenter l'exécution du programme à déboguer afin d'obtenir des informations sur la valeur de ses différentes variables durant son exécution. Au moment où une erreur se produit, on peut ainsi inspecter l'état du programme et en tirer des conclusions sur la cause du bogue. Les outils employant cette technique sont appelés des débogueurs *symboliques*. Ils sont capables de suspendre l'exécution du programme, pour inspecter les différents calculs en attente dans la pile d'exécution, ainsi que la valeur des variables locales ou globales. Ces outils permettent aussi de faire progresser l'exécution du programme « pas-à-pas », afin de visualiser les changements d'états qui ont lieu entre chaque pas. Conjointement à ces outils, on peut rajouter au code source des annotations appelées *assertions* ou *contrats*, qui servent à vérifier au moment de l'exécution qu'un certain nombre d'invariants dans le programme sont bien respectés. Si une assertion échoue, un message d'erreur indique l'endroit précis dans le code source où s'est produit le bogue ;
- *Le débogage par analyse de trace.* Cette approche consiste à enregistrer une trace d'événements — choisie selon le type de bogue que l'on cherche à corriger — qui se produisent tout au long de l'exécution, pour l'analyser ultérieurement. Cette approche est plus coûteuse que la précédente car elle ralentit l'exécution et peut nécessiter une grosse capacité de stockage pour conserver les données enregistrées. Néanmoins, elle est très utile lorsque les conclusions que l'on peut tirer de l'état courant d'un programme ne suffisent pas à déterminer l'origine d'un bogue. Certaines traces sont consultables à l'exécution. Par exemple, en traçant l'allocateur mémoire, on peut détecter que certaines parties du programme sont responsables de retentions de mémoire allouée pour des objets qui ne sont plus utilisés par la suite. D'autres types de traces sont utilisés afin d'obtenir des informations globales sur l'exécution d'un programme. Par exemple, dans le cas du débogage de performance, ou *profilage*, elle permettent de déterminer le pourcentage de temps global passé dans une fonction donnée. Les traces sont parfois le seul moyen de déboguer des programmes. C'est le cas par exemple avec les langages « paresseux » comme Haskell [PJAB+03], pour lesquels l'inspection de variables modifie l'évaluation du programme et donc son exécution.

Les techniques de débogage présentées ci-dessus sont complémentaires, en ce sens qu'elles permettent de tirer des conclusions différentes sur le programme analysé.

La technique la plus ancienne et la plus utilisée de nos jours reste le débogage symbolique. C'est sans doute la plus *naturelle* du fait de son interactivité. Elle est disponible pour pratiquement tous les langages de programmation et présente de grandes similitudes quel que soit le langage. Les travaux présentés dans ce document se concentrent sur ce débogage symbolique et plus précisément sur le débogage des langages de haut niveau compilés.

1.2 Les débogueurs symboliques et leurs limitations

Les débogueurs symboliques permettent d'instrumenter dynamiquement l'exécution d'un programme. Leur architecture et les services qu'ils fournissent dépendent principalement du fait que les programmes à déboguer soient interprétés ou compilés. Ces dernières années sont apparus des modèles hybrides mélangeant les caractéristiques des deux modèles précédents : ils sont basés sur du code-octet pouvant être compilé « à la volée » durant l'exécution. Les principales caractéristiques des deux modèles originaux sont détaillés ci-dessous.

1.2.1 Les débogueurs de programmes interprétés

Si un programme n'est pas transformé en langage machine pour s'exécuter, on dit qu'il est *interprété*. Dans ce cas, son exécution est prise en charge par un évaluateur fourni par l'implantation du langage dans lequel le programme est écrit. Le programme est rarement interprété sous sa forme symbolique (code source) pour des raisons d'efficacité. Il est d'abord compilé en un programme équivalent utilisant un langage *ad-hoc* propre à la plateforme d'exécution du langage. Les instructions de base du langage, appelées *code-octets*, sont de plus haut niveau que le langage machine et permettent d'encoder les abstractions du langage, comme par exemple les fermetures dans le cas des langages fonctionnels.

Les débogueurs de programmes interprétés sont dans la majorité des cas des outils intégrés dans la plateforme d'exécution du langage. Lorsque l'évaluation du programme provoque une erreur, l'exécution se suspend et le programme entre dans une boucle d'évaluation interactive, qui permet à l'utilisateur d'interagir avec l'interprète du langage pour visualiser les calculs en attente au moment de l'erreur, ou d'obtenir la valeur des variables présentes dans l'environnement de l'interprète à ce moment.

Les possibilités de contrôle de l'exécution varient grandement d'un débogueur à l'autre. Tous les débogueurs permettent de poser des *points d'arrêt* dans le code source des fonctions, afin de suspendre l'exécution lorsque ces endroits du code sont atteints. Dans certaines plates-formes d'exécution [LaL94], la pose de point d'arrêt s'effectue en ajoutant manuellement dans le code un appel à une fonction qui suspend l'exécution et entre dans une boucle d'évaluation interactive. Dans les plates-formes plus évoluées, il suffit de poser une marque à un endroit dans le code d'une fonction. Le code est ensuite automatiquement instrumenté [CFF01] pour mettre en œuvre les fonctionnalités de débogage.

Dans ce genre de modèle, le débogueur s'exécute dans le même espace mémoire que le programme à déboguer. Cela permet d'obtenir une forte interaction entre le débogueur, le débogué et l'utilisateur. Lorsque ce dernier est dans la boucle d'interprétation interactive du débogueur, il a accès à toutes les variables du programme. Il peut demander à l'interprète d'évaluer des expressions complexes, ou bien encore modifier son programme à la volée avant de reprendre son exécution.

1.2.2 Les débogueurs de programmes compilés

Dans ce modèle d'exécution, le débogueur s'exécute dans un processus indépendant, qui contrôle le processus du programme à déboguer (dans la suite, le *débogué*). Le débogueur doit donner une vision symbolique du code binaire en cours d'instrumentation. Pour y parvenir, lors de la compilation du programme, le compilateur produit des tables annexes qui contiennent les informations nécessaires au débogage :

- une table d'informations de lignes, pour mettre en correspondance le code binaire produit et le code source ;
- une table de symboles qui indique la position de chaque variable globale du programme dans le code binaire produit ;
- pour chaque fonction, une table de symboles qui établit une correspondance entre les variables locales d'une fonction et leurs positions dans les registres du processeur ou dans la pile d'exécution ;
- une table de descriptions des types, utilisée pour afficher correctement la valeur d'une variable en partant de sa représentation binaire à l'exécution.

L'instrumentation est externe. Il n'y a pas d'intrusion du débogueur dans le code source débogué. À l'exécution, le débogueur exploite les tables de correspondances pour présenter les informations sur l'état du programme à l'utilisateur. C'est une approche plus flexible que la précédente car il n'est plus nécessaire de modifier le code source.

Le débogueur contrôle l'exécution du programme « par ligne ». Il peut la suspendre lorsqu'une ligne particulière du code source est atteinte. L'exécution pas-à-pas s'effectue aussi par ligne. Cette granularité permet de visualiser aisément la progression de l'exécution dans l'environnement de programmation.

Le débogueur instrumente directement la forme compilée du programme, c'est-à-dire du code machine. Par conséquent, il ne dépend plus du ou des langages utilisés dans le code source du programme. Il est donc possible d'utiliser un seul outil pour déboguer plusieurs langages, ce qui facilite l'apprentissage par l'utilisateur.

1.2.3 Les limitations des modèles de débogueurs actuels

Tous les programmeurs sont un jour amenés à déboguer un programme ; il y a donc un réel besoin d'outils d'aide au débogage. Paradoxalement, ce domaine de l'informatique ne connaît plus d'évolution majeure : les programmeurs doivent se contenter des débogueurs existants, conçus il y a plus de vingt ans [Kat79, TD86, SP91] et dont les fonctionnalités ont globalement peu évolué. Ainsi, très souvent dans la pratique, ils préfèrent se passer complètement de ces outils au profit de solutions *ad-hoc*, consistant généralement à annoter le code source avec des `prints`. Cela impose de devoir recompiler le programme et de le relancer autant de fois que nécessaire pour localiser le bogue.

Ainsi, une question fondamentale se pose : « pourquoi utilise-t-on si peu les débogueurs malgré la difficulté manifeste du débogage et les limites des solutions *ad-hoc* ? ». Ce désintérêt s'explique sans doute par le fait que les outils dont on dispose ne sont pas pratiques à l'usage, trop limités ou qu'ils ne répondent pas, ou peu, aux besoins et aux attentes des utilisateurs. Essayons de détailler les principales limitations des débogueurs symboliques actuels.

Inconvénients des débogueurs pour programmes interprétés

Le support du débogage dans la plateforme d'exécution d'un langage est souvent réduit à la portion congrue. L'instrumentation de l'exécution est souvent obtenue par transformation de code source, qu'elle soit automatique [CFF01] ou manuelle [LaL94]. Cela entraîne souvent des problèmes de performances, ce qui interdit le débogage de gros programmes. De plus, cette approche par transformation de programme est délicate : il faut s'assurer que toutes les formes du langages soient transformées et que la transformation soit correcte, sous peine de changer la sémantique du programme [Kel95] et donc de fausser son débogage.

Contrairement aux débogueurs de langages compilés, de nombreuses de plates-formes d'exécution interprétées n'offre pas la possibilité de déboguer son programme « comme on l'édite », c'est-à-dire de visualiser la progression de l'exécution dans son éditeur de code. La seule interaction possible avec le débogueur est alors la ligne de commande qui, si puissante soit elle, ne remplace pas la visualisation intuitive fournie par l'éditeur. Cela est souvent dû au fait que les plates-formes ne savent pas faire l'association entre la représentation code-octet d'un programme et son code source. Les plates-formes les plus avancées fournissent l'intégration à l'environnement de développement [GR83], mais elles sont peu nombreuses.

Une des limitations fondamentales des débogueurs de langages interprétés est qu'ils ne permettent pas de déboguer un programme composé de plusieurs langages. En effet, un langage est pris en charge par sa propre plateforme d'exécution et les plates-formes utilisent toutes des représentations de code différentes. Il n'existe pas de pont entre les différents services de débogage que fournissent ces plates-formes, ce qui les rend mutuellement incompatibles.

Une conséquence directe des limitations précédemment décrites est qu'une plateforme d'exécution d'un langage ne sait pas instrumenter du langage machine. Or, dans la pratique, ce cas particulier de débogage multi-langage est très fréquent. Malheureusement, lorsqu'un programme utilise des bibliothèques pré-compilées, le débogage ne peut être au mieux que partiel. Le fait de devoir utiliser uniquement des bibliothèques interprétées est une contrainte si forte qu'elle rend le débogage bien souvent impossible dans la pratique.

Inconvénients des débogueurs pour programmes compilés

Le fait d'instrumenter directement les programmes au niveau du langage machine entraîne de nombreux problèmes liés à l'architecture matérielle. Cela rend difficile l'implantation de tels débogueurs et implique de fortes contraintes sur les utilisateurs.

Un débogueur doit fournir une instrumentation différente pour chaque architecture matérielle. En conséquence, son comportement varie d'une architecture à l'autre. Par exemple, l'exécution pas-à-pas diffère selon les processeurs. De plus, il existe plusieurs formats de table de débogage, à l'image des formats stabs [MKM93] et DWARF [Com95]. Cela handicape un peu plus la portabilité des débogueurs qui doivent supporter plusieurs formats de tables et implique que la qualité des services qu'il fournissent à l'utilisateur varie selon la précision des informations produites dans les tables.

Pour qu'un programme soit débogable, il est nécessaire de le compiler avec un ensemble d'options qui rendent son instrumentation praticable à l'exécution. Ce mode spécial de compilation implique plusieurs types de désagréments souvent rédhibitoires.

Premièrement, certaines optimisations classiques du compilateur ne peuvent plus être appliquées. Par exemple, on ne peut pas poser de point d'arrêt sur une fonction *inlinée*[1]. De même, l'allocation de registres doit tenir compte du débogage, sous peine de rendre l'inspection de la pile impossible sur certaines architectures matérielles. Ainsi, même si l'instrumentation du code machine ne ralentit pas l'exécution des programmes — le débogueur la met en œuvre en utilisant le support matériel fournit par le processeur [Pax90] — la chute de performance est proportionnellement beaucoup plus importante que celle observable dans les débogueurs de codes interprétés.

Deuxièmement, la compilation en mode débogage est une propriété *contaminante* : toutes les bibliothèques dont dépend le programme doivent être compilées pour le débogage, sous peine de limiter voire d'interdire le débogage du programme. Par exemple, certains compilateurs engendrent des champs synthétiques dans les structures des programmes lorsqu'ils sont compilés pour le débogage, ce qui les rend inutilisables pour des bibliothèques compilées sans support de débogage. Cette contrainte force les implanteurs de langages à maintenir deux versions différentes des bibliothèques qu'ils fournissent et force l'utilisateur à disposer des deux versions des bibliothèques. Cette lourdeur est bien souvent suffisante pour que les utilisateurs ne fassent pas l'effort d'utiliser un débogueur.

Les débogueurs de langages compilés peuvent déboguer un programme composé de

[1] Cette technique consiste à remplacer l'appel à une fonction par une copie de son corps.

plusieurs langages si ceux-ci sont réduits à du langage machine. Toutefois, les fonctionnalités de débogage qu'ils fournissent ont été conçues pour les langages dont les fonctionnalités et le modèle d'exécution se rapprochent de C, à l'image de Pascal, Ada ou Fortran. Hélas, ces débogueurs ne sont pas extensibles et ne permettent pas de déboguer les fonctionnalités des langages de haut niveau comme l'allocateur automatique de mémoire (ramasse miettes) ou les fonctions d'ordre supérieur. En particulier, ces débogueurs ne peuvent pas instrumenter les formes de code *ad-hoc* comme les fonctions interprétées, ce qui limite grandement leur efficacité.

1.2.4 Le problème du débogage des langages de haut niveau

Le vrai problème des débogueurs symboliques actuels est que ce sont des outils conçus pour déboguer des langages des années 70. Or, le débogage des langages compilés se complique dès lors qu'on considère les langages de haut niveau.

Ces dernières années ont connu une tendance vers l'unification des plates-formes d'exécutions des langages de haut niveau : les implanteurs de langages ont cherché des solutions de compilation qui leur permettraient de se passer de leur environnement d'exécution *ad-hoc* adaptés à leur langage, au profit d'une seule plateforme d'exécution généralisée. Cela a permis d'augmenter l'inter-opérabilité des langages et de leurs bibliothèques d'exécution. Parmi ces plates-formes généralistes, citons la Java Virtual Machine [LY97] et le .NET Common Language Infrastructure [GG01], deux machines virtuelles de haut niveau.

Le problème est que la compilation des langages de haut niveau vers une plateforme d'exécution généraliste comme la JVM est souvent complexe car les caractéristiques des langages ne correspondent pas aux fonctionnalités de la plateforme. Par exemple, les propriétés des structures de données des langages peuvent ne pas être compatibles avec celles supportées nativement par la plateforme d'exécution. De même, les langages peuvent disposer de constructions qui n'ont pas d'équivalent dans la plateforme d'exécution, comme par exemple les fermetures. Ces différences entraînent la production de structures de données synthétiques et d'appels de fonctions intermédiaires dans le code compilé. Cela a plusieurs conséquences du point de vue du débogage :

- l'inspection des fonctions peut faire apparaître des variables locales intermédiaires produites par le compilateur et qui ne sont pas présente dans le code source. De même, les structures de données utilisateurs peuvent contenir des champs synthétiques. Des types synthétiques produits pour implanter certaines fonctionnalités du langage peuvent aussi apparaître ;
- pour implanter certaines fonctionnalités, le compilateur peut avoir besoin de construire des fonctions synthétiques et de rajouter des appels de fonctions dans les fonctions utilisateurs originales. Cela pollue l'inspection de la pile d'exécution et perturbe grandement l'exécution pas-à-pas.

En résumé, la compilation délicate vers des plates-formes d'exécution généralistes n'est pas prise en compte par les débogueurs actuels. Ce manque de support spécifique interdit dans la pratique l'emploi d'un outil générique pour déboguer les langages de haut niveau et les programmes multi-langages.

1.3 Vers un meilleur débogage symbolique

Comme vu précédemment, les principales raisons du désintérêt des utilisateurs pour les débogueurs actuels peuvent être résumées comme suit :

- les débogueurs de code interprété sont intrinsèquement incapables de déboguer des programmes composés de plusieurs langages ou contenant du code compilé ;
- les débogueurs de code compilé imposent de lourdes contraintes sur la compilation des programmes. De plus, ils ne sont pas équipés pour déboguer efficacement les langages de haut niveau ;
- quel que soit le modèle, les débogueurs offrent un contrôle sur l'exécution et des fonctionnalités limités et ne sont malheureusement pas adaptables ou extensibles.

En conséquence, les débogueurs sont peu utilisés non pas parce que le modèle de débogage symbolique est mauvais — pour preuve, le débogage par `prints` ne fait que reproduire ce schéma — mais parce qu'ils sont trop contraignants.

La question que nous nous posons dans ces travaux est la suivante : comment améliorer les débogueurs symboliques pour les rendre plus attrayants ? Il est possible de répondre à cette question en suivant trois idées :

1. il est souhaitable d'avoir un outil unique basé sur le modèle des débogueurs de code compilé, car le modèle interprété ne permet pas de déboguer suffisamment de programmes « de tous les jours » ;
2. il faut réduire au maximum les contraintes d'utilisation (la compilation spéciale ou les limitations fonctionnelles) qu'implique l'architecture sous-jacente ;
3. il faut que l'outil puisse déboguer efficacement tous les aspects des langages de haut niveau (y compris leur interprète), ainsi que les programmes composés de plusieurs langages de haut niveau.

1.3.1 L'opportunité de la machine virtuelle Java

Avec l'arrivée du JDK 1.3 de la machine virtuelle Java (ou JVM), on dispose pour la première fois dans l'histoire du débogage symbolique de deux interfaces de programmation standard pour instrumenter une plateforme d'exécution : Java Debug Interface (JDI) et JVM Tool Interface[2] (JVMTI).

La JVM suit le modèle des débogueurs de code compilé : le débogueur et le débogué s'exécutent chacun dans leur propre JVM. JDI est une interface Java utilisée par le débogueur depuis sa JVM et qui permet d'instrumenter la JVM du débogué (par exemple poser un point d'arrêt, inspecter la pile d'exécution...). JVMTI est une interface C uniquement accessible depuis la JVM du débogué, qui permet de réagir à des événements de plus bas niveau, comme l'allocation dans le tas.

La plateforme JVM est très attrayante en ce qui concerne le débogage, car elle permet d'éviter toutes les contraintes dûes à l'architecture sous-jacente que l'on rencontre dans les débogueurs traditionnels :

- Les interfaces standards assurent la portabilité des débogueurs sur toutes les architectures supportées par la JVM et la cohérence des instrumentations et donc des fonctionnalités de débogage.

[2] JVMTI est en fait apparue dans le JDK 1.5. C'est une évolution de JVMDI et de JVMPI, précédemment apparues dans le JDK 1.3.

– L'implantation de l'instrumentation est déjà fournie, ce qui libère les implanteurs de débogueur d'une tâche fastidieuse.
– Il n'y a pas de compilation « spéciale débogage » du point de vue de l'utilisateur, puisque la compilation est effectuée à la volée au moment de l'exécution.
– Depuis le JDK 1.4, le compilateur à la volée (ou JIT) de la JVM reste activé pendant le débogage. Cela permet d'éviter des chutes de performances par rapport aux exécutions « normales ».
– Le JIT peut recompiler différemment [THL02] une fonction si les contraintes de débogage l'exigent. Par exemple, cela permet de conserver des optimisations comme l'inlining tout en gardant la possibilité de poser des points d'arrêt.
– En plus des fonctionnalités d'instrumentation fournies par JDI, JVMTI permet d'implanter des fonctionnalités avancées comme le débogage mémoire, ou d'ajouter des fonctionnalités à la volée selon le langage qu'on débogue (par exemple charger dynamiquement l'interprète du langage).

En résumé, la JVM libère les utilisateurs des contraintes matérielles (points 1 et 2 évoqués précédemment) et offre les outils nécessaires pour fournir de fonctionnalités de débogage à l'utilisateur.

1.3.2 Synthèse de la contribution de ces travaux

Dans ce document, il est proposé un ensemble d'améliorations à apporter au modèle de débogage basé sur l'instrumentation dynamique des programmes compilés afin de rendre les débogueurs plus efficaces et plus agréables à utiliser. Ces travaux se focalisent sur le problème de l'instrumentation d'un seul processus, pouvant contenir une ou plusieurs piles d'exécution (*multi-threading*). Ils ne prennent pas en compte le débogage *systémique*, qui consiste à instrumentation plusieurs processus simultanément (par exemple pour les programmes utilisant la migration). Ils ne prennent pas non plus en compte l'instrumentation du système d'exploitation. Le débogage de performances à l'exécution, ou *profilage*, n'est pas non plus abordé dans ces travaux.

Nos travaux ont conduit à l'implantation d'un débogueur appelé Bugloo [Cia03], distiné aux programmes compilés pour la JVM et composés d'un ou plusieurs langages de programmation. Voici ses principales caractéristiques :

– l'utilisation de la plateforme JVM permet d'éviter les contraintes de compilation spéciale des programmes. Cela simplifie grandement l'utilisation du débogueur ;
– il se contrôle à la ligne de commande à l'aide du langage fonctionnel Scheme [KCE98], ce qui permet de le scripter. Pour le rendre plus agréable à l'usage, il s'intègre à l'environnement de programmation GNU Emacs ou XEmacs ;
– contrairement aux débogueurs actuels, son architecture est modulaire et *extensible* : on peut facilement ajouter un support de débogage pour un nouveau langage, ou des fonctionnalités supplémentaires communes à plusieurs langages (par exemple le débogage de threads) ;
– c'est un débogueur entièrement *programmable* : une interface de programmation en Scheme expose ses fonctionnalités et ses capacités d'instrumentation. Cette interface est aussi accessible à la ligne de commande. Enfin, certaines fonctionnalités (par exemple les points d'arrêt) sont elles-même programmables, ce qui permet d'augmenter leurs expressivité.

Bugloo se singularise par sa capacité à déboguer des langages de haut niveau dont la compilation engendre des structures et des fonctions intermédiaires nuisibles à leur débo-

gage. Pour cela, des techniques de *représentations virtuelles* ont été développées dans le but de masquer la structure physique réelle des différentes abstractions du langage après leur compilation. Tout d'abord, le débogueur propose des mécanismes génériques d'affichage et d'inspection de l'état du débogué :

- un dispositif d'affichage de noms d'identifiants (fonction, variable, type...) encodés lors de la compilation pour pouvoir être représentables dans la JVM ;
- un mécanisme pour spécifier une ou plusieurs façons de formater les valeurs des abstractions fournies par un langage (fonctions, objets structurés...)
- un mécanisme générique permettant d'accéder aux différents niveaux de portées lexicales définies dans un langage et qui n'ont pas de contre-partie dans la JVM, par exemple les variables capturées ou globales ;
- un moyen d'inspecter des structures de données natives d'un langage qui doivent être émulées pour être représentables dans la JVM (listes, types énumérés...)

Le débogueur propose aussi un ensemble d'outils permettant de construire une représentation virtuelle de l'état du débogué dans le but de contrôler son exécution de manière transparente, quels que soient les langages utilisés dans le programme ou les particularités de leurs compilations :

- il peut construire une vue virtuelle de la pile d'exécution afin de masquer les appels de fonctions intermédiaires qui sont produits pour implanter les abstractions d'un langage. Une telle vue permet aussi de visualiser du code natif JVM et du code interprété dans une unique pile, de manière totalement transparente ;
- il peut faire du filtrage de saut durant l'exécution pas-à-pas afin de ne pas s'arrêter dans une fonction synthétique produite par le compilateur ;
- il fournit un moyen de poser des points d'arrêt dans des fonctions logiques — c'est-à-dire des fonctions définies dans le code source — même si elles n'ont pas été directement compilées en fonctions JVM.

Enfin, le débogueur offre des fonctionnalités de débogage mémoire, elles aussi indépendantes des langages utilisés et de la complexité de leur compilation :

- il peut produire des statistiques sur la quantité de mémoire utilisée dans le tas et la proportion pour chaque type du programme, aussi bien pour des types JVM que pour des types *ad-hoc* ;
- il fournit un inspecteur de références qui permet de retrouver l'origine de fuites mémoire ;
- il propose un profileur d'allocation mémoire adapté aux langages de haut niveau et à leur compilation complexe. Il construit des statistiques dans lesquelles les fonctions synthétiques produites par le compilateur sont remplacées par les fonctions utilisateur originales.

1.3.3 Le contexte de développement

Afin de valider les méthodes de prise en charge des langages de haut niveau développées dans ces travaux, un module de débogage complet a été réalisé pour permettre le débogage des programmes écrits dans le langage Bigloo [SW95], un dialecte du langage fonctionnel Scheme.

Traditionnellement les débogueurs de Scheme opèrent sur la forme interprétée des programmes, sans doute parce que ce langage se prête bien à la transformation de code. Dans ces débogueurs, la pile d'exécution est représentée par la liste des expressions en attente d'évaluation et la granularité de l'exécution pas-à-pas est l'expression.

Il peut paraître étonnant de choisir le modèle de débogueur de programmes compilés : en effet, à la différence de C, Scheme est un langage fonctionnel ; à ce titre, il ne différencie pas les expressions des instructions. Par conséquent, le flot de contrôle des programmes Scheme est souvent plus « perturbé » que celui des programmes C. Cependant, contrairement aux langages paresseux comme Haskell qui rendent l'utilisation des débogueurs symboliques très difficile (cf. les travaux de Ennals [EJ03]), les programmes Scheme sont aisément débogables, car leur évaluation suit une progression séquentielle.

Le compilateur Bigloo dispose de plusieurs générateurs de code. Ainsi, il peut produire du code C, du code-octet JVM ou du code-octet .NET. Bigloo fournissait déjà un débogueur nommé BDB [SB00] pour les programmes compilés en C, mais il a été abandonné. Il était trop contraignant à utiliser car il nécessitait une compilation spéciale pour le débogage. De plus, il utilisait le débogueur GDB [SP91] pour instrumenter les programmes et cette connexion était trop difficile à maintenir. Enfin, il ne permettait pas de déboguer certains aspects du langage comme l'interprète. Une des raisons pour concevoir un générateur de code-octet JVM était de pouvoir bénéficier d'un débogueur comme Bugloo qui ne souffre pas de ces limitations.

Le langage Bigloo est significatif car il présente beaucoup de fonctionnalités de haut niveau : fonctions d'ordre supérieur, capture de variables, capture du flot d'exécution avec des continuations, macros, interprète de code, structures de données *ad-hoc*, etc... Chacune de ces fonctionnalités requiert une compilation assez complexe. De plus, le langage dispose d'un système de *threads* particulier, dans lequel des fils d'exécutions partagent l'espace mémoire du programme et s'exécutent de manière synchrone [SBS04]. En résumé, les caractéristiques de ce langage permettent d'illustrer un grand nombre de mécanismes mis en œuvre dans un débogueur qui a vocation à être générique et extensible.

1.4 Organisation de ce document

La suite de ce document est organisée de la manière suivante : le chapitre 2 présente un état de l'art sur le débogage. Le reste de la thèse est découpé en deux grandes parties.

La première partie du document décrit les outils et les techniques de représentations virtuelles développés pour ces travaux. Le chapitre 3 présente les principales fonctionnalités du débogueur Bugloo et les mécanismes génériques d'inspection et d'affichage de valeurs mis en œuvre pour lui permettre de déboguer correctement les langages de haut niveau à la compilation complexe. Le chapitre 4 décrit une nouvelle technique de construction de vue virtuelle de pile permettant de masquer les détails de compilation présents dans la pile d'exécution et de fournir une exécution pas-à-pas correcte quel que soit le langage de haut niveau utilisé. Le chapitre 5 présente les fonctionnalités de profilage mémoire et leur application aux langages de haut niveau à la compilation complexe.

La seconde partie de ce manuscrit est consacrée à l'illustration des mécanismes d'extensions et de représentations virtuelles fournis par le débogueur Bugloo. Le chapitre 6 décrit les extensions développées pour supporter le débogage des programmes Bigloo. C'est un exemple concret du travail qui doit être effectué par les implanteurs souhaitant utiliser Bugloo. Le chapitre 7 présente le débogage d'un programme multi-langage afin de montrer que les techniques de représentations virtuelles développées dans Bugloo fonctionnent en présence de plusieurs langages. Le chapitre 8 présente une extension permettant de déboguer le système de *threads* de Bigloo. C'est un exemple d'utilisation de l'API du débogueur pour étendre ses fonctionnalités.

Enfin, le chapitre 9 conclut la thèse et présente des perspectives de travaux futurs.

Chapitre 2

État de l'art

Il existe deux manières d'appréhender le débogage des programmes. La première consiste à rechercher un maximum d'erreurs avant l'exécution à l'aide d'outils spécifiques. La seconde consiste à constater un bogue à l'exécution puis à relancer le programme en l'instrumentant pour rechercher la cause du bogue. Ces deux techniques sont strictement complémentaires car elles permettent d'exhiber des types de bogues différents. Ce chapitre dresse un panorama des techniques de débogage existantes dans les deux catégories. Il conclu en présentant l'approche de débogage retenue dans ces travaux et la contribution qu'ils apportent.

2.1 Le débogage par analyses statiques

Les outils de débogage statique utilisent les propriétés logiques des langages de programmation dans lesquels sont écrits les programmes pour en tirer des conclusions logiques. Ces conclusions permettent de déterminer si un type de bogue est susceptible de se produire à l'exécution. La suite de cette section décrit les différentes techniques de débogage statique.

2.1.1 Outils et analyses classiques

L'un des plus vieux outil de débogage statique est sans doute Lint [Joh79], un testeur de programmes C. Il permet aux programmeurs d'être rigoureux sur l'utilisation du système de type de C ou de détecter les variables inutilisées. Il analyse aussi le flot de contrôle pour vérifier la bonne utilisation des `return`, `break` ou `continue`.

Utilisées dans de nombreux outils, les analyses ensemblistes [HJ94] permettent de fixer un domaine de définition aux différentes variables du programme. En propageant ces informations dans le programme, les analyses peuvent détecter des valuations de variables pour lesquelles l'exécution provoquerait une erreur. Des variantes de ces outils [FF96] permettent d'analyser les programmes de manière modulaire et peuvent donc traiter de plus gros programmes.

Les analyses de pointeurs [EGH94, Hin01] essaient de déterminer quels sont les zones mémoires dans le programme qui sont susceptibles d'être référencées par un pointeur. Cette information peut ensuite être utilisée pour déterminer les pointeurs du programme qui peuvent accéder aux mêmes zones mémoires. Ces analyses d'alias sont centrales afin de déterminer si des zones mémoires sont accédées ou modifiées.

Certaines analyses proposent de détecter certains types d'erreurs ou certaines propriétés indésirables en les modélisant à l'aide de systèmes de types. Vérifier qu'une propriété est valide dans le programme revient alors à vérifier qu'elle est typable. Par exemple, Flanagan et Freund proposent une analyse utilisant un système de types [FF00a] pour détecter des *race conditions* dans les programmes Java. De même, Matos et Boudol [MB05] proposent un système de types et d'effets pour vérifier que des informations « sensibles » provenant d'une partie d'un programme ne puissent pas être inférées dans une autre partie.

Le découpage du programme en tranches logiques, ou *program slicing* [Wei82] permet de déterminer statiquement toutes les parties du programmes qui peuvent avoir un effet sur une variable du programme. Cette technique donne des indications à l'utilisateur sur les parties du programme à inspecter pour détecter la source d'un bogue.

2.1.2 Assertions et contrats pour le débogage

Les assertions sont des expressions logiques qui permettent d'exprimer des propriétés sur un programme. Le débogage statique de ces assertions est très vieux. Par exemple, Syntox [Bou93b, Bou93a] est un débogueur qui utilise des techniques d'interprétations abstraites [CC77] d'un sous-ensemble du langage Pascal pour déterminer statiquement certains comportements d'un programme à l'exécution. Dans Syntox, les assertions sont essentiellement utilisées pour décrire des invariants de programmes de petite taille.

La programmation par contrat [Mey92a] a été popularisé avec le langage Eiffel [Mey92b]. Elle consiste à annoter les fonctions de son programme avec des assertions exécutées avant (pré-condition) et après (post-condition) la fonction utilisateur. Par exemple, on peut utiliser les contrats pour exprimer le fait que la valeur d'une variable de type entier doit être comprise entre 0 et 100. En ce sens, les contrats peuvent être vus comme un complément de typage.

JML [LBR99] est un langage de spécification comportemental pour Java. Pour construire des assertions, il fournit un langage proche de Java étendu par des quantificateurs universels pour l'expressivité de la logique. Les travaux de Trentelman et Huisman [TH02] étendent le langage d'assertion de JML pour fournir des opérateurs de la logique temporelle.

2.2 Le débogage durant l'exécution

Le débogage dynamique permet d'instrumenter l'exécution d'un programme d'un point de vue « bas niveau ». Pour pouvoir inspecter l'état d'un programme, il faut souvent outrepasser la sémantique du langage de programmation utilisé dans ce programme. Par exemple, un débogueur efficace doit pouvoir inspecter les champs privés des objets structurés du langage, même si la sémantique l'interdit. La suite de cette section présente un panorama des techniques de débogage à l'exécution.

2.2.1 Débogage pour environnements interprétés

Les débogueurs conçus pour les environnements interprétés sont en général des interprète particulier pouvant être contrôlé afin d'instrumenter l'exécution du programme. Dans ce modèle, l'utilisateur peut toujours accéder aux variables de son programme à travers l'interprète, ce qui permet un débogage très interactif.

Smalltalk et ses dérivés

Le langage Smalltalk [Sho79, GR83] fournit dans les années 70 ce qui est sans doute l'ancêtre des débogueurs symboliques modernes. Il continue encore de nos jours à influencer les débogueurs, comme par exemple le débogueur générique d'Eclipse[1].

Le débogueur Smalltalk est un interprète spécialisé de code-octet Smalltalk qui interrompt le programme lorsque ce dernier provoque une erreur. Il peut aussi être invoqué par un appel de fonction placé dans le code source pour suspendre le programme à un point précis. Lorsque le programme est suspendu, le débogueur donne différent types d'informations :

- il affiche le contenu de la *pile d'exécution*, c'est-à-dire la liste des message Smalltalk en attente d'exécution ;
- il permet d'inspecter la liste des variables locale pour chaque fonction présente dans la pile ;
- il permet de continuer d'exécuter le programme instruction par instruction ;
- il permet de changer le corps d'une fonction « à chaud » et de poursuivre l'exécution du programme débogué ;

En Smalltalk, le débogueur est un simple programme, ce qui le rend aisément programmable : l'utilisateur peut construire des scripts servant à étendre son comportement ou ses fonctionnalités. De même, il est possible de « déboguer le débogueur ».

Self [US87] est un langage à prototype descendant de Smalltalk. Dans la machine virtuelle Self, le code octet interprété est dynamiquement traduit en code natif optimisé. Le débogueur peut fournir des services similaire à celui de Smalltalk grâce à sa capacité à remplacer les remplacer les fonctions optimisées présentes dans la pile par leur contre-partie interprété (en code-octet Self).

VisualWorks [How95] est un environnement de programmation Smalltalk-80. En plus des fonctionnalités de débogage précédemment citées, son débogueur dispose de point d'arrêt conditionnels permettant de suspendre le programme en fonction de la valeur d'une expression Smalltalk. Il permet aussi de déboguer des parties du programme source en les sélectionnant avec la souris dans l'éditeur intégré. Enfin, il peut inspecter les champs des structures de données de manières graphiques et afficher les pointeurs reliant ces structures dans la mémoire. « Squeak [IKM+97] est un environnement Smalltalk graphique dont la machine virtuelle est elle-même écrite en Smalltalk. Le débogueur intégré peut instrumenter ou inspecter n'importe quelle partie du programme, mais aussi n'importe quelle partie de l'environnement graphique intégré. D'autre débogueur sont disponibles, à l'image d'Unstuck [HDD06] : cet outil trace l'exécution du programme pour pouvoir examiner les différentes valeurs qu'ont pris les variables avant la survenue d'un bogue. »

Les débogueurs Lisp

Les environnements de développement Lisp ont très tôt proposé une variété d'outil de débogage. Ils se présentent en générale sous la forme d'interprète spécialisés. Par exemple, l'environnement InterLisp-D offre différents outils capable d'instrumenter l'évaluation du programme :

- il propose un traceur qui permet d'afficher à l'écran les entrées ou les sorties de fonctions durant l'exécution ;

[1] Darin Wright, responsable du développement de cet outil, était auparavant impliqué dans le développement de l'environnement de programmation ENVY/Smalltalk.

- lorsque le programme est suspendu suite à une erreur inattendue, il permet de visualiser la liste des expressions symboliques en attente d'évaluation, ainsi que les différentes variables de l'environnement d'évaluation ;
- il offre la possibilité de remplacer une expression en cours d'évaluation par une valeur arbitraire puis de continuer l'exécution ;

Certains environnements comme LeLisp [CDH84] fournissent des *steppers*, outils permettant une évaluation pas-à-pas du corps des fonctions : chaque étape de l'évaluation des s-expression est ainsi détaillé à la ligne de commande.

Il existe d'autres outils d'exécution pas-à-pas, moins verbeux que les steppers. ZStep [Lie84] représente l'évaluation d'une fonction sous une forme symbolique, dans laquelle les variables sont substituées par leur valuation au cours de l'exécution. Son successeur, ZStep'95 [LF95] construit représentation graphique animées représentant les différentes phases de l'évaluation des programmes Lisp.

Lisp est un langage représentant le code et les données de la même manière. Ainsi, les outils de débogage fonctionnent souvent en effectuant de la transformation automatique de code. Par exemple, PSD [Kel95] est un débogueur qui manipule le code source du programme pour fournir les fonctionnalités de trace ou d'exécution pas à pas.

2.2.2 Débogage pour environnements compilés

Les débogueurs symboliques de programmes compilés ont un modèle d'exécution différents des débogueurs pour environnements compilés. Ils s'exécutent dans un processus et instrumentent un autre processus dans lequel s'exécute le code machine du programme à déboguer. L'utilisateur accède aux zones mémoires du programme débogué de manière « symbolique », c'est-à-dire en employant les noms des variables du programme. Pour cela, le programme doit être compilé dans un mode spécial afin de produire des tables de débogage dans le fichier binaire final.

La dichotomie inhérente à ce modèle d'exécution réduit grandement l'interactivité des sessions de débogage. Toutefois, les différents débogueurs symboliques ont su évolués pour offrir des services se rapprochant des débogueurs symboliques pour les environnement interprétés.

ADB

Les premiers débogueurs de code compilé étaient très rudimentaires, à l'image du débogueur ADB [MB77]. Cet outil permet d'inspecter des images mémoire de programmes terminés inopinément à la suite d'un bogue. Il offre aussi un environnement dans lequel l'exécution d'un programme peut être surveillée et contrôlée. Ce débogueur est basé sur l'inspection de code machine et d'adresses mémoire. Les seules informations de débogage dont il se sert sont les labels, c'est-à-dire l'adresse des fonctions et des variables en mémoire. En conséquence, les fonctionnalités de débogage qu'il propose sont très limitées :

- il permet de poser des points d'arrêt, mais uniquement sur le début des fonctions. L'exécution se suspend dès que l'une de ces fonctions est appelée ;
- il permet de visualiser les fonctions en attente dans la pile d'exécution. La position à l'intérieur d'une fonction est exprimée par un décalage en octet par rapport à son début, ce qui est très « bas niveau » ;
- il permet d'inspecter des zones mémoire, notamment les positions utilisées par les variables locales ou globales. Les valeurs sont présentées sous forme décimale ou hexa-

décimale, il n'y a pas de facilité pour inspecter des objets structurés de manière « symbolique ».

SDB

SBD [Kat79] est l'un des premiers débogueurs à considérer le programme à déboguer au niveau de son *code source*. C'est un débogueur pour le langage impératif C. Cet outil utilise des informations de débogage supplémentaires produites par le compilateur C, comme les informations de lignes ou les informations symboliques sur les types structurés du programme. Les fonctionnalités de débogage sont donc étoffées :

- il permet de poser des points d'arrêts dans les fonctions dans n'importe quelle ligne du code source ;
- il permet de visualiser les fonctions en attente dans la pile d'exécution, en affichant la ligne dans le code source à laquelle ces fonctions sont suspendues ;
- il fournit un moyen d'exécuter le programme *pas-à-pas*, grâce aux informations de ligne ;
- il permet d'inspecter les champs des objets structurés du programme ;
- il a la capacité d'appeler des fonctions dans le programme débogué lorsque l'exécution est suspendue.

DBX

Conçu à l'origine pour le langage Pascal, le débogueur DBX [TD86] a vite évolué pour permettre le débogage des langages couramment utilisés à l'époque (C, Pascal, Fortran, Modula...) et a été porté sur un très grand nombre d'architectures matérielles. DBX dispose des mêmes fonctionnalités de contrôle et d'inspection que SDB, mais fournit de nouveaux types de points d'arrêts :

- les points d'arrêt mémoire ou *watchpoint*, permettent de suspendre l'exécution lors de l'accès à une zone mémoire du programme ;
- les points d'arrêt temporaires permettent d'activer ou de désactiver un point d'arrêt après un certain nombre de passages ;
- les points d'arrêt conditionnels, qui sont associés à une expression du langage[2], suspendent l'exécution uniquement lorsque l'évaluation d'une expression associé au point d'arrêt renvoie la valuer *vrai*.

En plus de ces points d'arrêt, DBX fournit un langage de commande « ad-hoc » permettant de configurer le débogueur ou de créer des macros pour appeler plusieurs commandes consécutivement.

DBX est la première incarnation des débogueurs symboliques modernes. Ses descendant, comme GDB [SP91], JDB [CMM+97] ou JSWAT [Fie99], sont tous des adaptations fournissant peu ou prou les mêmes fonctionnalités.

2.2.3 Évolution des débogueurs symboliques

Déboguer en voyageant dans le temps

Certains débogueurs permettent de « remonter le temps » : le programmeur peut rejouer dans le détail une partie de l'exécution du programme plutôt que de relancer le débogueur.

[2]Le langage des expressions est en réalité un sous-ensemble du langage utilisé dans le code source du programme.

Dans leur débogueur pour SML [TA95], Tolmach et Appel permettent de poser des points de passage (ou *checkpoints*) dans le temps, continuer l'exécution et revenir en arrière ultérieurement. Son fonctionnement repose sur l'instrumentation de code source à la compilation. Durant l'exécution, l'état du débogué est capturé au moyen d'une *continuation*. Pour conserver l'état des variables mutables, le code source est instrumenté de manière à mettre à jour une table de hachage globale avant chaque opération d'écriture.

BugNet [NPC05] fournit un mécanisme de retour dans le temps en sauvegardant à intervalles réguliers le minimum d'information pour être capable de rejouer les instructions machines précédent la survenue d'un bogue. Cet outil vise à rejouer du code utilisateur et pas les appels systèmes. Il fonctionne en sauvegardant la valeur des registres machines au début d'un point puis en traçant les instructions machines de chargement de valeur dans les registres. Malheureusement, il nécessite un support matériel dédié.

Omniscient Debugger [Lew03] propose d'enregistrer tous les changements d'état se produisant durant l'exécution d'un programme Java. Un fois l'exécution terminée, la trace enregistrée peut être parcourue pour émuler une exécution pas-à-pas, pour revenir dans le temps, pour suivre un déclenchement d'exception, un changement de contexte... Toutefois, l'implantation de ce genre de technique sature vite la mémoire vive et peut ralentir l'exécution du programme d'un facteur 300 !

Inspection graphique de données

Les débogueurs symboliques peuvent être complétés par des outils d'affichage graphique de l'état du programme débogué.

GDBX [Bas85] est une extension de DBX qui s'emploie pour afficher le contenu des types structurés ou des tableaux à une ou plusieurs dimensions. Les objets inspectés sont affichés dans une fenêtre en deux dimensions et des flèches symbolisent les pointeurs qui les relient entre eux. L'utilisateur peut modifier les objets structurés ou les pointeurs à l'aide de la souris.

DDD [ZL96] étend l'idée de visualisation graphique de GDBX. Cet outil s'utilise en amont (c'est un *frontend*) des débogueurs standards comme DBX, DGB ou JDB. Il utilise un langage proche de TeX appelé VSL pour construire les graphes. Ses heuristiques d'affichage sont extensibles par l'utilisateur.

2.2.4 Débogage par analyse de traces

Débogage systémique

Le débogage systémique est une sorte de débogage utilisé pour instrumenter l'ensemble des programmes en cours d'exécution au lieu d'instrumenter un unique processus. Ce type d'outil permet de retrouver des bogues causés par une mauvaise interaction entre différents processus, y compris le noyau du système d'exploitation.

Dtrace [CSL04] est un débogueur systémique conçu pour l'environnement UNIX Solaris, capable d'instrumenter dynamiquement les programmes utilisateurs et le noyau. Il permet de définir des points d'instrumentation dans les processus vivants dans le système et d'exécuter du code à chaque fois qu'un de ces points est atteint. Cet outil peut réagir aux exécutions d'appels systèmes dans le noyau (plus de 30000 points d'entrée) et à n'importe quel exécution de fonction utilisateur. Il peut aussi réagir au changements d'état des données des processus, comme par exemple le changement d'état d'un *thread* ou d'une pile d'exécution.

Pour utiliser Dtrace, l'utilisateur écrit des scripts en langage D qui seront appelés à chaque passage sur un point d'instrumentation. Ces scripts sont automatiquement compilés en langage machine et sont injectés « à chaud » dans les processus durant leur exécution. Grâce à la bibliothèque de scripts prédéfinis, l'utilisateur peut récolter des statistiques sur l'exécution de ses programmes, par exemple le nombre d'appels systèmes, la taille maximum de la pile ou le nombre d'octet alloués.

Des débogueurs systémiques similaires à Dtrace commencent à voir le jour, notamment sous le système Linux avec systemtap [PCEH05].

2.2.5 À la frontière du débogage : le profilage

Le profilage sert à obtenir des statistiques sur les ressources consommées durant l'exécution d'un programme.

Gprof [GKM82] est probablement le premier profileur servant à mesurer le temps passé dans les fonctions d'un programme C. Pour fonctionner, le programme doit être compilé dans un mode « spécial profilage » qui rajoute dans le prologue et l'épilogue des fonctions du code nécessaire à l'instrumentation. Le profileur fonctionne par échantillonage : il détermine la fonction en sommet de pile à intervalle régulier, la gratifie du temps écoulé et fait de même pour les fonctions en attente dans la pile. Dans Gprof, le temps total imputé aux fonctions en attente repose sur une approximation qui peut se révéler incorrecte.

Les profileurs modernes imposent moins de contraintes d'utilisation. C'est le cas par exemple pour OProfile [Lev04], un profileur de temps d'exécution, récemment développé pour le système Linux. Ses fonctionnalités sont similaires à Gprof, mais elles fonctionnent en instrumentant dynamiquement le programme à l'exécution. Il n'est donc pas nécessaire d'utiliser un mode de compilation spécial. Le profileur JVM hprof [O'H04] fonctionne aussi par instrumentation, mais exige de lancer la machine virtuelle dans un mode spécial profilage.

Outils spécialisés

Certains outils de débogage spécialisés permettent de tracer l'exécution des programmes pour localiser des types de bogues difficilement détectables avec des débogueurs symboliques classiques.

DejaVu [ACN+01] est un débogueur capable de rejouer de manière déterministe les exécutions non-déterministes des programmes Java *multi-thread*. Il fonctionne avec la machine virtuelle Java Jalapeño[3]. Cette JVM fournit un ordonnancement quasi-préemptif à l'aide de *timers* et de points de coopérations insérés dans le programme par le compilateur. Le débogueur peut ainsi enregistrer les changements de contexte qui ont lieu à l'exécution.

JProfiler [Et04] est un outil servant à analyser la cause d'un verrou mortel, ce blocage collectif survenant à la suite d'une mauvaise exclusion mutuelle. Il est capable d'enregistrer l'activation des différents *threads* d'un programme JVM et les prises de moniteurs qu'ils effectuent. Il fonctionne par échantillonage en déterminant à intervalles réguliers le *thread* actif. Lorsqu'un verrou mortel se produit, l'utilisateur peut analyser le cheminement qui a mené au verrou.

[3] Récemment renommée Jikes RVM.

2.2.6 Autres approches de débogage à l'exécution

Il existe d'autres types d'outils de débogage à l'exécution qui diffèrent des débogueurs symboliques ou des outils de trace.

Instrumentation de code machine

Certaines techniques de débogage fonctionnent par instrumentation du code machine. Bien que délicates à réaliser, ces transformations permettent de glaner de nouvelles informations très utiles au débogage.

Purify [HJ91] est une sorte de contrôleur d'allocation mémoire. Il est capable de détecter l'accès à des zones mémoires non initialisées ou des appels à `malloc` et à `free` dépareillées. Il peut aussi détecter la présence de fuites mémoires, ces zones mémoires allouées qui ne sont pas rendues au système. Pour fonctionner, purify instrumente le code machine du programme avant son exécution.

Valgrind [NS03] est un méta-outil de mise au point de programme. Les programmeurs l'utilisent pour construire des outils pouvant superviser de très nombreux aspects de l'exécution d'un programme, comme par exemple toutes les fonctionnalités de purify, un profileur de mémoire cache ou encore un détecteur de corruptions mémoires (*data races*). Valgrind est très dynamique, il fonctionne en émulant l'exécution du programme dans son propre processeur virtuel. Il emploie des techniques de compilation à la volée ou JIT pour conserver des performances acceptables.

Débogage événementiel

Les outils offerts par les débogueurs symboliques pour contrôler l'exécution sont en nombre limités. Souvent, il serait souhaitable de pouvoir contrôler plus finement l'exécution en fonction de l'état précis de son programme.

Dalek [OCH91] est un débogueur programmable construit au dessus de GDB. Il produit des événements en réaction à des passages dans certains points du programme. Chaque événement est associé à une fonction de rappel (écrite dans un langage *ad-hoc* proche de C) qui peut elle-même produire d'autres événements. Cela simule un flot de données qui permet de contrôler finement le programme débogué.

Les travaux de Marceau et Cooper [MCKR04] étendent l'idée de débogage événementiel. Leur modèle utilise un moteur programmation réactive asynchrone [CK03] pour diffuser les événements à travers le graphe de flot de donnée. Le langage utilisé pour les fonctions de rappel est un Scheme réactif embarqué dans l'environnement de programmation DrScheme [FCCF+02], ce qui permet de construire des fonctions de débogage très expressives.

2.3 Approche retenue dans ces travaux

Les sections précédentes ont présenté un éventail de techniques pour la mise au point de programmes. Ces techniques peuvent être classées en deux grandes catégories : l'approche statique et l'approche dynamique. Dans ces travaux, l'approche dynamique a été retenue, car les débogueurs symboliques sont un moyen intuitif et efficace de contrôler l'exécution de ses programmes.

Les débogueurs symboliques modernes conçus pour les programmes compilés proposent tous des fonctionnalités similaires à celles de DBX. Le principal inconvénient à l'égard de

ces techniques est qu'elles ne sont pas adaptées au débogage des langages de haut niveau « modernes » comme les langages fonctionnels ou les langages scriptables :

- soit parce qu'elles sont conçues pour un langage particulier, comme c'est le cas par exemple pour Dalek ;
- soit parce qu'elles ne tiennent pas compte des spécificités des langages de haut niveau, notamment de leur compilation délicate vers les plates-formes d'exécution généralistes.

Dans tous les cas, les fonctionnalités des débogueurs symboliques modernes ne sont pas conçues pour être étendues en fonction du langage source utilisé dans les programmes utilisateur. De plus, ils ne permettent pas de déboguer efficacement des programmes composés de plusieurs langages de haut niveau.

Le solution développée dans ces travaux est de concevoir un débogueur symbolique programmable. Ce débogueur doit permettre aux utilisateurs de personnaliser le débogueur à leur convenance. De plus, les fonctionnalités de débogage qu'il propose doivent être utilisables pour les langages de haut niveau. Enfin, le débogueur doit pouvoir déboguer efficacement les programmes composés de plusieurs langages de haut niveau. La suite de cette thèse décrit les solutions apportées pour répondre à ces besoins.

Première partie

Un débogueur générique

Chapitre 3

Le débogueur Bugloo

Ce chapitre présente une vue d'ensemble du débogueur Bugloo dont la particularité est de permettre un débogage efficace des programmes multi-langages et de tenir compte de la compilation complexe des langages de haut niveau vers des plates-formes d'exécution généralistes comme la JVM. Le débogueur expose une interface de programmation qui permet aux implanteurs de langages de personnaliser le comportement des fonctionnalités de débogage comme la pose de point d'arrêt, la recherche de variable locale ou l'exécution pas-à-pas. Il met également en œuvre des mécanismes de filtrage permettant de masquer les détails de compilation polluant l'inspection, à l'image des variables dont le nom a été encodé ou des structures contenant des champs synthétiques. La suite du chapitre décrit les fonctionnalités générales du débogueur et les différentes manières d'étendre ses fonctionnalités grâce à son interface de programmation. Nous décrivons ensuite les mécanismes génériques mis au point pour interpréter et afficher correctement la valeur des objets provenant des langages de haut niveau. Enfin, nous décrivons l'implantation de Bugloo et en particulier l'utilisation des interfaces JDI et JVMDI dans le débogueur et leur impact sur les performances du programme débogué.

3.1 Présentation du débogueur

Bugloo est un débogueur de programmes compilés s'exécutant dans la JVM. Son architecture est basée sur le modèle de débogage des programmes compilés : le débogueur s'exécute dans une JVM et instrumente une seconde JVM dans laquelle s'exécute le programme débogué. L'utilisateur contrôle le débogueur au moyen d'un langage de commande intégré. Bugloo peut être utilisé dans un terminal ou peut s'intégrer à l'environnement de programmation. Dans ce dernier cas, le programmeur dispose d'une interface unique pour éditer et mettre au point ses programmes. Cette section décrit le déroulement d'une session de débogage et présente les principales fonctionnalités du débogueur ainsi que leur intégration dans l'environnement de programmation GNU Emacs.

3.1.1 L'environnement de débogage Bugloo

Bugloo dispose d'une interface complète à l'environnement de programmation GNU Emacs [Sta81]. Cette interface utilise Bee [Ser00], un environnement de programmation multi-langages pour Emacs. Le démarrage du débogueur se fait depuis le menu d'une fenêtre contenant du code. Les figures 3.1 et 3.2 présentent des copies d'écran d'une session de

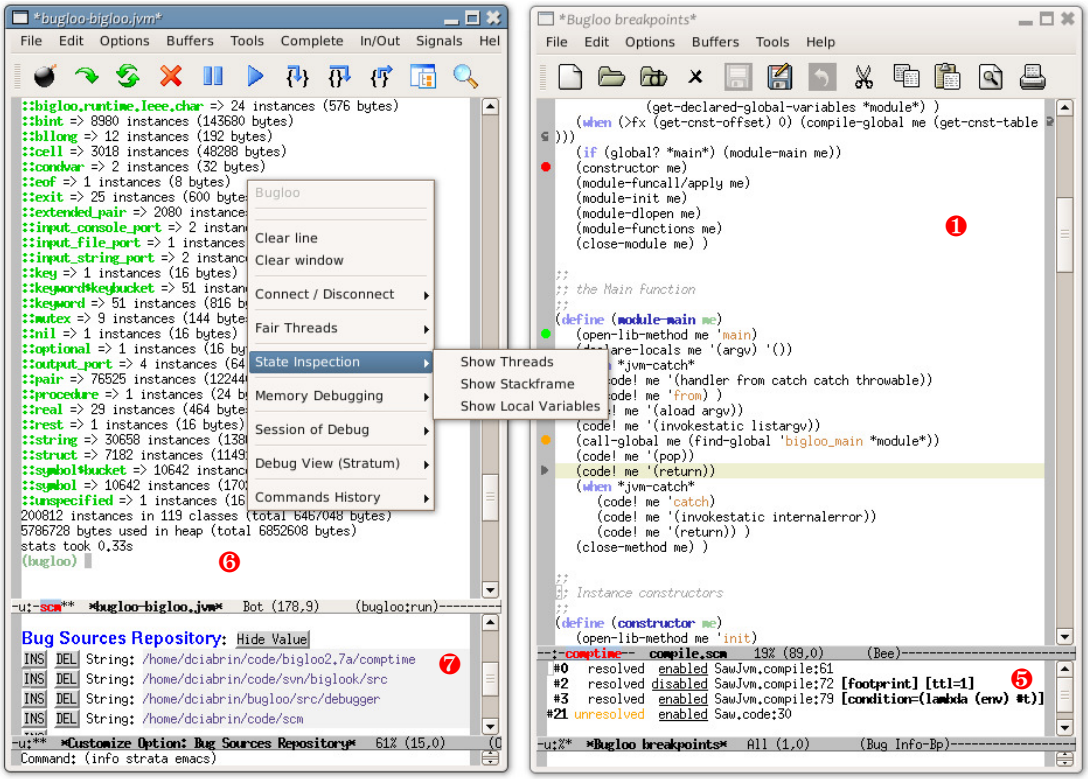


FIG. 3.1: Session de débogage dans l'environnement de programmation

débogage. L'utilisateur interagit avec le débogueur à l'aide de la ligne de commande, de la barre d'outils ou d'un menu déroulant. Les fonctionnalités principales disposent de leurs propres fenêtres :

❶ La fenêtre de **code source** interagit avec le débogueur lorsqu'elle est *connectée*. La marge gauche d'une fenêtre est plus épaisse pour indiquer sa connexion. L'utilisateur peut alors se servir de la souris dans cette marge pour créer, activer ou supprimer des points d'arrêt dans le code source. La couleur des points d'arrêt indique leur état d'activation. Lorsque l'exécution est suspendue, la position correspondante dans le code source est automatiquement soulignée et une flèche s'affiche dans la marge.

❷ La fenêtre de **threads** affiche la liste des *threads* s'exécutant dans le débogué, ainsi que l'état dans lequel ils se trouvent à un moment donné de l'exécution. Le *thread* actif au moment de la suspension est affiché en gras. Cliquer sur un *thread* permet d'afficher sa pile d'exécution dans la vue ❸.

❸ La vue de la **pile d'exécution** permet d'inspecter les fonctions en attente dans un *thread*. Lorsqu'une fonction est sélectionnée, la position du flot de contrôle à l'intérieur de cette fonction est soulignée dans la fenêtre de code (❶). De plus, la liste des paramètres et des variables locales de la fonction est affichée dans la vue ❹.

❹ La vue des **variables locales** présente le nom, le type et la valeur de chaque argument ou variable locale défini dans la fonction courante. L'interface utilise la fonctionnalité Emacs de *tag* [Sta81], servant à indexer les définitions présentes dans un fichier de code source. Ainsi, lorsque l'utilisateur clique sur le type d'une variable, sa définition est automatiquement affichée dans la vue ❶.

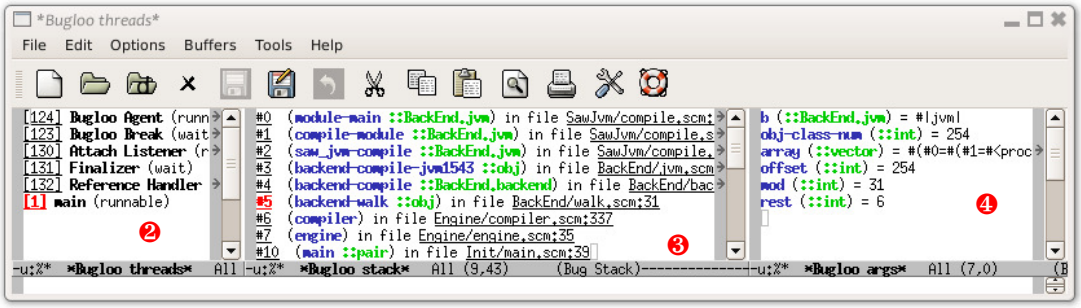


FIG. 3.2: Inspection de l'état du programme débogué

❺ La fenêtre des **points d'arrêt** donne des informations sur l'état des différents points d'arrêts. Chaque ligne indique la position d'un point d'arrêt dans le code, si ce point d'arrêt est actuellement actif et s'il est déjà effectif, c'est-à-dire si la classe contenant le code à suspendre a déjà été chargée). Les différents attributs (cf. section 3.3) du point d'arrêt, comme la durée de vie (`ttl` dans la figure) ou la condition d'arrêt, sont présentés entre crochets.

❻ La **ligne de commande** permet à l'utilisateur d'interagir avec BUGLOO manuellement. Il peut accéder aux fonctionnalités du débogueur non exposées dans l'interface graphique, ou bien personnaliser le débogueur en construisant ses propres macro-fonctions.

❼ Le **dépôt de code source** maintient un ensemble de chemins sur disque contenant les fichiers sources des différents programmes à déboguer. Cela est nécessaire car l'information de fichier source présente dans une classe JVM contient en général un chemin relatif.

3.1.2 L'inspecteur structurel

La connexion entre BUGLOO et Emacs permet de visualiser simplement l'état des différentes variables locales ou globales du programme débogué. Lorsque l'utilisateur souhaite visualiser la valeurs des champs d'un objet structuré, il peut utiliser l'inspecteur graphique de BUGLOO. Cet outil utilise Biglook [GS03a], une bibliothèque de construction d'interfaces graphiques pour Scheme. Cela permet de s'affranchir des limites imposées par l'interface textuelle de l'éditeur Emacs.

Un inspecteur construit pour l'utilisateur une représentation graphique — appelée une *vue* — de l'état d'un objet présent dans le programme débogué. La vue utilisée par l'inspecteur est choisie en fonction du type de l'objet. Les implanteurs de langages ou de bibliothèques peuvent utiliser l'API de BUGLOO pour programmer des extensions fournissant de nouvelles vues.

La figure 3.3 est une copie d'écran présentant un inspecteur structurel. À la base de la fenêtre graphique, la barre de statut indique le type de l'objet en cours d'inspection. Au milieu de la fenêtre réside la vue choisie pour représenter l'objet en cours d'inspection. Dans une vue, les champs pointant sur d'autres objets peuvent être eux-mêmes inspectés dans la fenêtre d'inspection courante ou dans une nouvelle. La barre d'outil au sommet de l'inspecteur fournit des fonctionnalités communes à toutes les vues :

– quand l'utilisateur inspecte un nouvel objet dans la fenêtre courante, la vue représen-

FIG. 3.3: Capture d'écran d'un inspecteur structurel

tant l'ancien objet est conservée dans un historique des inspections. Il est possible de se déplacer dans cet historique, à la manière des navigateurs Web. Quand une nouvelle vue est créée, elle est insérée à la position courante dans l'historique et seules les vues précédentes sont conservées.

– un objet particulier peut être représenté par différentes vues. Par exemple, si l'objet inspecté provient d'un langage de haut niveau, sa structure peut contenir des champs laissant apparaître des détails de compilation (comme des champs synthétiques). Une vue spéciale peut lui être associée afin de masquer ces détails. Le dernier bouton de la barre d'outils permet de choisir la vue à utiliser pour l'inspection. Quel que soit le type de l'objet inspecté, le débogueur fournit une vue par défaut (figure 3.3) présentant une introspection basique de l'objet.

3.2 Contrôle par le langage de commande

Le débogueur est contrôlable au moyen d'un langage de commande. Contrairement à de nombreux débogueurs [TD86, SP91, Fie99, BKO+02] contrôlable au moyen d'un langage *ad-hoc* souvent limité, le langage de commande fourni par BUGLOO est le langage Scheme. Cela permet à l'utilisateur de « personnaliser » le débogueur avec toute la puissance d'un vrai langage. Il peut, par exemple, créer des macros-commandes en utilisant des fermetures ou des variables globales.

BUGLOO étant programmé en Scheme, toute l'API d'instrumentation du débogué est facilement accessible à partir de la ligne de commande. Par exemple, l'utilisateur a la possibilité de charger des classes JVM à la volée dans le programme débogué et d'appeler des fonctions distantes. De plus, les objets que retournent ces fonctions peuvent être manipulés depuis la ligne de commande comme s'il s'agissait d'objet « locaux ». Le fait d'utiliser une API unique pour programmer la ligne de commande et pour ajouter des fonctionnalités de débogage complexes comme le support des FairThreads (cf. chapitre 8) est important : cela permet une grande liberté de personnalisation tout en offrant un modèle de programmation homogène.

Le mécanisme de la ligne de commande diffère de la boucle de lecture-évaluation-affichage classique de Scheme. En effet, BUGLOO maintient un historique de toutes les commandes lues depuis le clavier durant une session de débogage. Cet historique peut

être sauvegardé sur disque et rechargé ultérieurement. L'intérêt de cet enregistrement est double :

- il permet de sauver de manière « rudimentaire » l'état du débogué et la configuration courante du débogueur. L'utilisateur peut ainsi arrêter une session de débogage pour la reprendre ultérieurement. Il peut aussi « rejouer » une session sans avoir à retaper toutes les commandes. Bien sûr, cette fonctionnalité ne fonctionne pas correctement sur des programmes non-déterministes, mais elle reste pratique dans de nombreux cas.
- le chargement d'historique est utilisé comme mécanisme de configuration. Au démarrage de BUGLOO, un historique « système » est évalué pour initialiser le débogueur. Cet historique peut être modifié par les implanteurs de langages afin d'initialiser les fonctionnalités de débogage propres à leur langage. Cela leur permet par exemple de créer des points d'arrêt par défaut à chaque nouvelle session, de rajouter des vues à l'inspecteur graphique ou bien de définir des filtres fournissant une vue virtuelle de la pile d'exécution (cf. chapitre 4).

La ligne de commande BUGLOO évalue continuellement des expressions Scheme. L'exécution d'une commande BUGLOO s'apparente à un appel de fonction classique, mais dans lequel les arguments ne sont pas évalués[1]. Par exemple, la commande suivante pose un point d'arrêt dans la fonction `fun` de la classe `foo` :

```
(bugloo) (bp add foo fun)
```

Les arguments sont passés à la commande `bp` sous leur forme symbolique. L'utilisateur peut forcer la ligne de commande à évaluer des arguments en utilisant la forme `unquote` de Scheme :

```
(bugloo) (bp add foo ,(+ 10 20))
```

La commande précédente pose ainsi un point d'arrêt dans la classe `foo` à la ligne 30.

3.3 Instrumentation du flot de contrôle

Comme tous les débogueurs, BUGLOO permet de poser des points d'arrêt dans différents points logiques de l'exécution du programme débogué afin de suspendre son exécution :

- sur passage à un endroit donné dans le code source ;
- sur lecture ou écriture d'un champs d'un objet structuré ;
- sur le déclenchement d'une exception ;
- sur l'entrée ou le retour d'une fonction.

Contrairement aux autres débogueurs, il n'existe pas dans BUGLOO de type particulier pour représenter des variantes des points d'arrêts précédents, comme par exemple les points d'arrêt temporaires ou conditionnels. Pour le débogueur, ces variantes sont considérées comme des *attributs*.

Dans BUGLOO, les attributs sont des fonctions prenant un point d'arrêt en paramètre et retournant un booléen. Ils peuvent être associés à n'importe quel type de point d'arrêt précédemment évoqué. Dans le débogueur, ils sont représentés par des mot-clés Scheme, suivis par leurs arguments éventuels :

```
(bugloo) (bp add foo bar :ttl 1 :footprint)
```

[1] le choix a été fait de ne pas évaluer les arguments des commandes BUGLOO afin de faciliter l'utilisation de la ligne de commande.

Le point d'arrêt précédent est posé dans la classe `foo` au début de la fonction `bar`. Lorsqu'un point d'arrêt est atteint, tous ses attributs sont exécutés en séquence. Si l'un des attributs renvoie la valeur *faux*, l'action de suspension du point d'arrêt est invalidée et l'exécution du programme reprend. Différents attributs sont disponible par défaut dans le débogueur :

ttl la durée de vie d'un point d'arrêt. À chaque passage sur le point d'arrêt, sa durée est décrémentée. Lorsqu'elle atteint zéro, l'attribut commande au débogueur de supprimer le point d'arrêt et renvoie la valeur `faux` pour reprendre l'exécution.

footprint cet attribut permet d'afficher un message à chaque fois qu'un point d'arrêt est atteint avant de reprendre aussitôt l'exécution. C'est un moyen de tracer l'exécution sans avoir à insérer de `print` et recompiler le programme.

thread cet attribut active le point d'arrêt seulement si le *thread* qui a déclenché la suspension correspond à l'argument associé à l'attribut.

trace cet attribut enregistre le nom de la fonction se trouvant en sommet de pile avant de relancer l'exécution. Il peut être utilisé avec des points d'arrêt sur entrée de fonction pour obtenir la liste des fonctions appelées dans une partie particulière du programme. Ce type de trace est lent mais néanmoins utile dans certains cas.

emacs cet attribut est utilisé lorsque le débogueur est démarré depuis Emacs. Lorsqu'un point d'arrêt est atteint, l'attribut envoie un code spécial sur la sortie standard pour forcer l'éditeur à actualiser ses fenêtres de code source. Comme cet attribut a une vocation uniquement informative, il renvoie toujours `vrai`.

L'attribut **custom** est un attribut générique qui permet d'exécuter une fonction personnelle, dans le but de modéliser des comportements supplémentaires. Par exemple, il est possible de rendre effectif un point d'arrêt après un certain nombre de passages de la manière suivante :

```
(bugloo) (bp add foo bar
          :custom ,(let ((n 10))
                     (lambda ()
                       (or (<= 0 n) (begin (set! n (- n 1)) #f) ))))
```

Les implanteurs de langages peuvent créer leur propres attributs additionnels à l'aide de l'API de programmation de Bugloo. Ce type d'approche est beaucoup plus efficace que celle employée dans les débogueurs traditionnels car elle est extensible et elle est applicable à n'importe quel type de point d'arrêt. De plus, elle multiplie les types de points d'arrêt que peut construire l'utilisateur, plutôt que de le limiter à un nombre prédéfini de combinaisons, comme cela est le cas dans les débogueurs traditionnels.

Hormis le mécanisme de sauvegarde d'historique, le débogueur ne prévoit pas de moyen direct de sauver la liste des points d'arrêt présents dans le débogueur pour les réutiliser dans une session ultérieure. Toutefois, l'utilisateur peut définir sa propre fonction. Cette fonction peut utiliser l'API de programmation de Bugloo pour retrouver la liste des points d'arrêt et en sauver une représentation sur disque. Dans ce cas, c'est à lui de s'assurer de la validité des valeurs sauvées. En particulier, la fonction associée à l'attribut **custom** ne peut pas être sauvegardée car, lors de l'appel à la commande `bp`, la forme symbolique de cette fonction n'est pas conservée et ne peux donc plus être recréée.

3.4 Affichage virtuel pour les langages de haut-niveau

La compilation des langages de haut niveau vers des plates-formes généralistes produit en général diverse structures de données intermédiaires nécessite souvent un encodage des identificateurs présents dans le code source. Pour déboguer de tels programmes, il faut savoir interpréter le contenu des informations de débogage et prendre en considération les spécificités des langages de programmation durant l'inspection. La suite de cette section décrit les différents mécanismes mis au point dans Bugloo pour y parvenir, quel que soit le ou les langages de programmation utilisés dans le programme à déboguer.

3.4.1 Le système d'informations de lignes en strates

La compilation des langages de haut niveau ne produit pas forcément directement du langage machine pour la plateforme d'exécution. Dans de nombreux cas, la compilation produit un code équivalent au code original mais dans un langage de plus bas niveau, proche de la plateforme d'exécution. Cela permet de déléguer la phase de production de code natif à un compilateur déjà existant. Dans le cas de la JVM, le langage intermédiaire est souvent Java.

L'inconvénient de la compilation intermédiaire est que les informations de lignes présentes dans l'exécutable final décrivent le code source intermédiaire au lieu du code original. Si certains langages intermédiaires permettent de contourner cette limitation (par exemple, pour le langage C, à l'aide de la directive `#line` du pré-processeur), cela reste impossible en utilisant le langage Java.

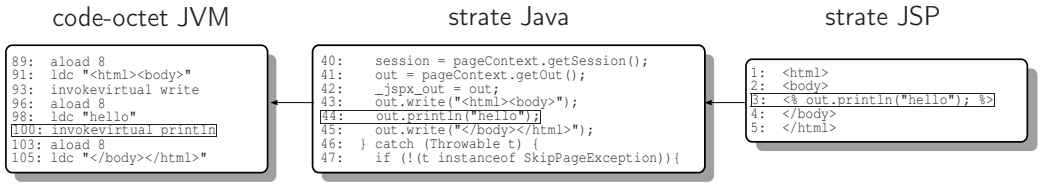


Fig. 3.4: Informations de lignes pour les langages intermédiaires

Depuis le JDK1.4, le format des classes JVM a été étendu pour permettre de conserver une table d'informations de lignes par langage intermédiaire utilisé durant la compilation d'un code source. Pour illustrer cette extension, la figure 3.4 représente un programme JSP compilé en Java puis en code-octet. Dans un tel programme, les informations de débogage sont réparties en *strates* représentant chacune un langage source différent. La strate Java associe du code octet JVM à du code source. Chaque strate supplémentaire doit être définie en fonction de la précédente. Une ligne dans une strate (ici JSP) peut représenter une ou plusieurs lignes dans la strate précédente (ici Java), mais pas l'inverse.

Bugloo permet d'effectuer un débogage en strate à l'aide d'un mécanisme extensible. Chaque implanteur de langage déclare les noms des strates qu'il construit dans les classes JVM. Durant une session de débogage, l'utilisateur peut sélectionner une strate courante pour chaque langage enregistré auprès du débogueur. Lorsque l'exécution est suspendue, le débogueur détermine la classe JVM d'où provient la fonction en sommet de pile. Puis il effectue un traitement lui permettant de retrouver le langage source utilisé pour le corps de la fonction. Ce mécanisme est détaillé dans le chapitre 4. Lorsque le langage est déterminé, sa strate courante est retrouvée. Le débogueur utilise ainsi les informations de ligne de cette

strate. En particulier, la strate utilisée fixe la granularité de l'exécution pas-à-pas, comme le montre la figure 3.4.

La bonne interprétation des informations de lignes est nécessaire, mais elle ne garantit pas que l'exécution pas-à-pas ou l'inspection de pile soit exempte d'artefact dû à la compilation complexe des langages de haut niveau. Le chapitre 4 présente les techniques mises au point pour traiter spécifiquement ce problème.

3.4.2 Mécanismes génériques d'inspection du programme

Lors de l'inspection d'un programme utilisant un ou plusieurs langages de haut niveau, le débogueur doit avoir des connaissances sur les schémas de compilation de ces langages pour pouvoir être plus facile à utiliser : il doit connaître la fonction d'encodage et de décodage des identificateurs ; il doit pouvoir afficher la valeurs des variables de la même manière que la bibliothèque d'exécution ; lors de la recherche d'une variable, il doit pouvoir parcourir tous les niveaux lexicaux définis par les langages ; il doit inspecter correctement les structures de données utilisateur ayant subi des transformations durant la compilation.

Les mécanismes d'inspection mis au point dans BUGLOO sont génériques. Ils peuvent être étendus par programmation et fonctionnent en présence de multiples langages. La suite de cette section décrit leurs caractéristiques.

Décodage des identificateurs et affichage des objets

Les informations de débogage présentes dans une classe JVM contiennent des identificateurs représentant le nom de différents types d'objets définis dans le code source, comme par exemple des noms de variables, des noms de structures ou de types, des noms de fonctions, etc... Dans le cas des langages de haut niveau, il est fréquent que la phase de compilation encode les identificateurs avant de construire les informations de débogage. Cette pratique est généralement employée lorsque les identificateurs du langage contiennent des caractères qui ne sont pas autorisés dans la JVM ou lorsque le langage fournit son propre mécanisme d'espace de nommage, différent de celui de la JVM.

BUGLOO propose un mécanisme de décodage générique dans lequel chaque langage de programmation est traité par un *module* particulier fourni par l'implanteur. Un module est chargé du décodage des identificateurs, comme les noms de fonctions, de classes, de variables locales ou de champs de structures. Il est aussi responsable du formatage de la signature des fonctions pendant l'inspection de la pile. Cela permet par exemple pour les langages comme Lisp de conserver un affichage « préfixe ». Enfin, le module doit être capable d'encoder des identificateurs pour permettre à l'utilisateur de poser des points d'arrêts sur des noms de fonctions.

Lors de l'inspection, chaque identificateur est automatiquement formaté par le module prenant en charge l'affichage de son langage. Pour cela, le débogueur demande successivement à chacun des modules s'il peut décoder l'identificateur. Un module par défaut pour le langage Java traite les identificateurs non pris en charge par les autres modules. Il se contente de les afficher « tels quels ».

En plus des décodeurs, BUGLOO fournit un mécanisme générique d'afficheur pour formater de différentes manières la valeur des objets d'un langage. Cela permet d'utiliser la fonction de formatage par défaut du langage. Cela peut aussi servir à fournir plusieurs fonctions d'affichage pour un langage si sa bibliothèque le permet. Les implanteurs de langages doivent fournir un ou plusieurs modules d'affichage qui seront associés à leur module

de décodage. Durant la session de débogage, l'utilisateur peut fixer le module d'affichage courant pour chaque module de décodage.

Par défaut, le module de décodage (langage Java) est complété par un module d'affichage utilisant la fonction Java standard `toString` : lorsque la valeur d'un objet du programme débogué doit être affichée, le débogueur fait un appel distant dans le programme débogué afin d'exécuter la fonction `toString` de l'objet, puis affiche le résultat. Un exemple complet d'afficheurs est présenté au chapitre 6.

Rechercher une variable dans le programme

Lorsque l'utilisateur veut afficher la valeur d'une variable ou d'un champs de structure du programme, il donne son nom au débogueur afin que ce dernier recherche l'objet correspondant dans le débogué. Lorsque le nom dénote une variable locale, il suffit de chercher le nom parmi les variables locales du bloc d'activation courant. Sinon, la recherche dépend du langage source utilisé, c'est-à-dire des différents niveaux syntaxiques ou lexicaux dont il dispose. À titre d'exemple, dans le cas du langage Bigloo, la recherche doit prendre en compte l'environnement des variables capturées et celui des variables définies dans l'interprète.

Dans BUGLOO, la recherche de variable est un mécanisme configurable par des extensions fournies par les implanteurs de langage. Une extension connaît les différents environnements syntaxiques du langage qu'elle prend en charge, ainsi que les détails de sa compilation vers la JVM. Elle est donc capable de rechercher l'identificateur demandé par l'utilisateur dans tous les environnements syntaxiques du langage qui sont accessibles depuis le bloc d'activation courant. Pour sa part, le débogueur n'a pas besoin de connaître les spécificités des langages de haut niveau. Son rôle se limite à déterminer le langage source utilisé dans le bloc d'activation courant, puis à déléguer la recherche de variable à l'extension prenant en charge ce langage.

Durant une session de débogage, plusieurs extensions peuvent coexister afin de supporter le débogage des programmes multi-langage. Par défaut, une extension est automatiquement chargée pour le support du langage Java. Si le bloc d'activation courant représente une méthode d'instance, l'extension recherche le nom demandé par l'utilisateur parmi les champs de cet instance. Si l'objet n'est pas trouvé, la recherche se poursuit dans les champs statiques de l'instance. Au cas où la classe de l'instance serait une classe interne, la recherche est étendue au champs statiques de toutes les classes englobantes.

Représentation virtuelle des objets structurés

Dans la JVM, le seul type structuré natif est la classe. Les langages de haut niveau disposent eux aussi de leur propres type structurés. Lorsque ces types sont compilés ils sont représentés par des classes JVM.

Lorsque les caractéristiques des types structurés diffèrent trop de celles des classes JVM, la compilation introduit généralement des champs intermédiaires dans les classes de sortie afin d'encoder les spécificités des types originaux. Dans ce cas, un simple mécanisme de décodage des nom de champs (cf. 3.4.2) ne suffit plus pour inspecter correctement ces structures : le débogueur doit être capable de masquer les détails de compilation apparents. Son travail varie selon le schéma de compilation employé :

– Lorsque le type à compiler est « semblable » à une classe JVM, sa compilation ne rajoute tout au plus que quelques champs synthétiques. Par exemple, la compilation des objets Jython [Hug97] crée des champs statiques contenant des constantes et des

informations de débogage. Dans ce genre de cas, il suffit que le débogueur masque ces champs synthétiques ;
- Lorsque le type a des caractéristiques trop éloignées d'une classe JVM, l'implanteur de langage doit maintenir un système de type *ad-hoc* en parallèle avec celui de la JVM. Par exemple, le langage ECMAScript [ECM99] fournit un système à objet à base de prototypes. Le compilateur Rhino [Fou98] représente un objet ECMAScript par un objet JVM intermédiaire. Cet objet a deux champs synthétiques qui permettent de conserver le nom du prototype et une table d'association contenant tous les champs de l'objet original et leurs valeurs associées. Lorsque le débogueur inspecte un tel objet, il doit afficher son type original et uniquement présenter ses champs originaux ;

BUGLOO fournit un moyen générique de construire une représentation virtuelle des objets présentés durant l'inspection de l'état du programme débogué. Dans l'API BUGLOO, les objets du programme débogué peuvent être manipulés à travers des objets du débogueur de type `dbg-object`. Comme pour la recherche de variable, le mécanisme de représentation virtuelle est configurable au moyen d'extensions chargées au démarrage de la session de débogage. Ces extensions doivent sous-classer `dbg-object` afin de fournir une objet spécialisé capable de renvoyer des valeurs virtuelles pour toutes les caractéristiques de l'objet distant à manipuler : type, parent, champs, valeurs... Les objets virtuels peuvent être obtenus en utilisant le mot-clé `:filter` à la ligne de commande :

```
(bugloo) (info args :filter)
```

Dans l'exemple ci-dessus, le débogueur retourne une vue virtuelle des variables locales du bloc d'activation courant. Pour chaque variable représentant un objet JVM, le débogueur demande aux extensions chargées si une représentation virtuelle de l'objet doit être construite. Ce sera le cas si l'objet contient des détails de compilation apparents, comme des champs synthétiques. Chaque extension peut ainsi s'occuper des types d'objets produits par le langage qu'elle prend en charge et, le cas échéant, retourner un objet spécial si l'objet JVM expose des détails de compilation qui doivent être masqués pour l'utilisateur. Enfin, le mot-clé `:filter` peut aussi être employé en conjonction avec l'inspecteur structurel (cf. section 3.1.2).

3.5 Implantation

Cette section décrit le fonctionnement interne du débogueur. Elle présente les différentes API de débogage mises à disposition par la JVM pour construire des outils de débogage. Nous décrivons ensuite les mécanismes mis en place au dessus de ces API pour implanter les différentes fonctionnalités de débogage de BUGLOO. Enfin, une rapide étude de performances présente l'impact du débogueur sur l'exécution des JVM déboguées.

3.5.1 APIs de débogage de la Machine Virtuelle Java

L'architecture d'une session de débogage JVM implique l'interaction entre deux machines virtuelles. Le débogueur s'exécute dans la première et il est chargé d'établir une connexion avec la seconde JVM. Pour cela, il peut :
- démarrer une seconde JVM dans un mode spécialement prévu pour le débogage. Dans ce mode, certaines optimisations du JIT sont débrayées pour permettre un débogage du code assemblé ;

– s'accrocher dynamiquement à une JVM en cours d'exécution. Pour cela, il faut que cette JVM ait été préalablement démarrée dans le mode spécial débogage.

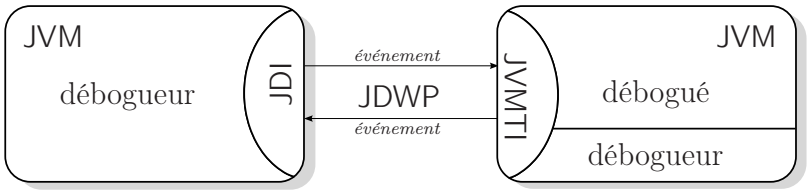


FIG. 3.5: Architecture opérationnelle d'une session de débogage

La figure 3.5 présente un schéma de l'architecture de débogage de la JVM. La communication entre le débogueur et le débogué est spécifiée par l'API standard JVM Tool Interface (JVMTI). Cette interface de programmation définit un ensemble de *requêtes* que le débogueur peut envoyer à la JVM instrumentée afin d'obtenir des informations sur l'état du programme débogué durant son exécution. L'interface définit aussi des *événements* que la JVM instrumentée envoie en réponse à des changements d'état interne comme le passage sur un point d'arrêt, ou l'entrée dans une fonction.

Les requêtes et les événements JVMTI sont transmis grâce au protocole JDWP (Java Debug Wire Protocol). Ce protocole définit un canal de communication abstrait entre les deux JVMs. Plusieurs types de connexions bas-niveau sont disponibles : tubes UNIX, mémoire partagée ou sockets BSD. Cela permet de déboguer un programme distant ou local de manière transparente.

Dans la JVM du débogueur, les informations JVMTI sont accédées à travers une interface de haut niveau Java Debug Interface (JDI). Cette interface réifie les concepts de base comme une pile d'exécution, un bloc d'activation ou un point d'arrêt, pour les utiliser aisément depuis le langage Java. De plus, elle conserve en cache les informations sur les types inspectés, ce qui augmente les performances de l'inspection.

Certaines fonctionnalités de débogage doivent être exécutées directement dans la JVM du débogué pour éviter des transferts d'information entre les JVMs qui nuirait aux performances. Pour cela, il est nécessaire de charger dans la JVM du débogué un *agent de débogage* natif qui pourra directement utiliser l'API JVMTI pour l'instrumentation. Le chapitre 5 présente un exemple d'une telle fonctionnalité.

3.5.2 L'architecture du débogueur Bugloo

L'API JVMTI fournit les « briques » pour faire communiquer deux JVMs entre elles. Il est nécessaire de construire d'autres fonctionnalités au dessus de celles fournies par cette API pour réaliser un débogueur complet.

La figure 3.6 est une vue de haut niveau de l'architecture de BUGLOO. Le débogueur est conçu pour réagir aux interactions de l'utilisateur depuis l'interface graphique et depuis la ligne de commande et des événements provenant du débogué. Pour cela, il est implanté à l'aide de Fair Threads [SBS04], une bibliothèque de *threads* coopératifs synchrone. Dans ce modèle, l'exécution est divisée en unités de temps logiques appelées *instants*. À chaque instant, tous les Fair Threads sont exécutés. Les Fair Threads communiquent entre eux en attendant ou en émettant des *signaux* diffusés instantanément à tous les Fair Threads.

Dans l'architecture, le débogueur s'exécute à l'aide de la boucle d'événements de l'interface graphique. Lorsque l'interface est au repos, le thread graphique joue le rôle d'un

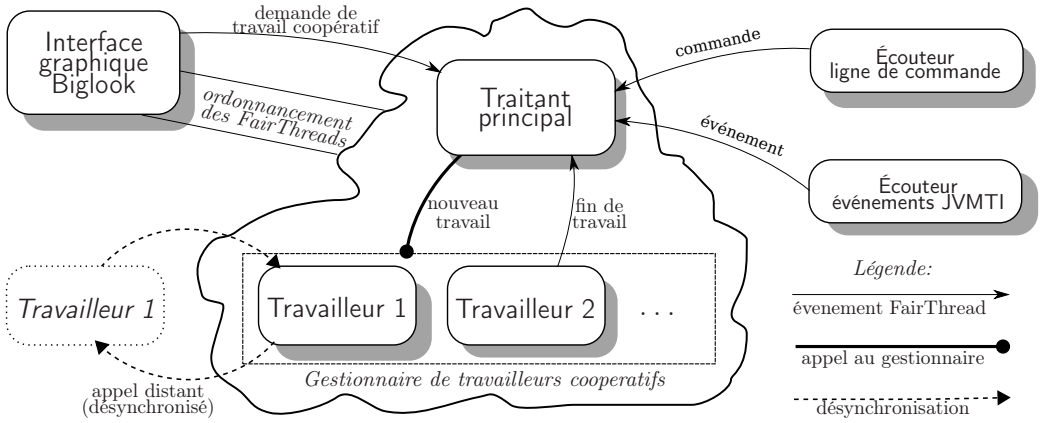


FIG. 3.6: Le cœur du débogueur et sa gestion des événements extérieurs.

ordonnanceur et exécute des instants tant que des nouveaux événements sont diffusés par les Fair Threads. Des *threads* JVM indépendants écoutent la ligne de commande et la connexion au débogué. Dès que des commandes ou des événements peuvent être traités, les threads émettent un signal dans la zone de Fair Threads et enregistrent une tâche future dans la boucle d'événements qui ordonnancera les Fair Threads.

Lors d'un ordonnancement, le Fair Thread principal est réveillé par un signal et traite l'événement correspondant. Pour cela, il choisit un *travailleur* libre parmi un *pool* de Fair Thread et lui demande d'effectuer le traitement (par exemple, afficher les variables locales). Si le travailleur doit effectuer un appel distant dans la JVM du débogué (par exemple, pour obtenir la valeur d'un objet), il doit se désynchroniser le temps de l'appel car un appel distant JVMTI est bloquant. De cette manière, le débogueur peut continuer à écouter les événements JVMTI sans bloquer en attendant la fin de l'appel[2].

3.5.3 Traitement des événements provenant du débogué

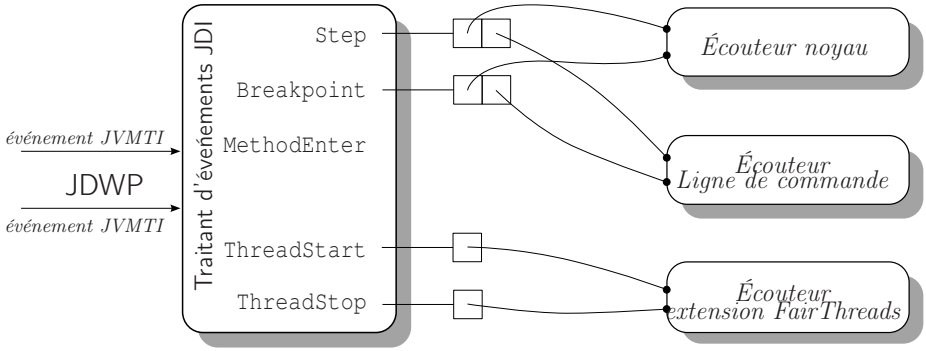


FIG. 3.7: Le traitement d'un événement JVMTI par les écouteurs.

Les événements JVMTI transmis du débogué jusqu'au débogueur indiquent un changement d'état de la JVM du débogué. On reçoit des événements pour notifier :

[2]En particulier, cela évite un verrou mortel au cas où l'appel de fonction distant passe par un point d'arrêt.

– le démarrage ou la terminaison de la JVM ;
– l'avancement d'un pas durant l'exécution pas-à-pas ;
– le chargement d'une classe JVM ;
– la levée d'une exception JVM ;
– la création ou la mort d'un *thread* ;
– le passage sur un point d'arrêt ;
– l'entrée ou la sortie d'une fonction JVM.

La figure 3.7 illustre le traitement de ces événements par le débogueur. Ce dernier a une architecture extensible qui permet de diffuser les événements aux différentes extensions chargées au démarrage de la session de débogage. Chaque événement est associé à une liste d'écouteurs à notifier :

– l'écouteur « noyau » est toujours enregistré. Il met en œuvre les fonctionnalités principales du débogueur, comme l'évaluation des attributs des points d'arrêt ;
– un autre écouteur enregistré à la suite du premier sert à afficher des informations à la ligne de commande, comme le bloc d'activation courant lors du passage sur un point d'arrêt ;
– des écouteurs supplémentaires peuvent être installés par les extensions du débogueur. Dans la figure 3.7, un écouteur additionnel est installé par l'extension du débogage des Fair Thread (cf. chapitre 8) pour mettre à jour la liste des *threads* lorsque l'exécution se suspend.

La séparation des écouteurs permet de « court-circuiter » la propagation des événements pour les masquer. Par exemple, lorsque l'écouteur noyau a déterminé qu'un point d'arrêt temporaire n'est plus valide, il court-circuite la propagation pour éviter que l'écouteur secondaire n'affiche un message d'arrêt à l'écran.

3.5.4 Extensibilité grâce aux appels de fonctions distants

Évoluer avec deux JVM séparées permet de s'assurer que le débogueur n'influe pas trop sur l'évolution normale du programme débogué. Toutefois, il est parfois impératif d'exécuter du code dans la JVM déboguée. Par exemple, Bugloo a besoin de charger des classes dans le débogué pour fournir certaines fonctionnalités de débogage mémoire. La suite de cette section décrit le mécanisme de chargement de classe et d'appels de fonctions dans la JVM du débogué. Elle présente aussi les précautions à prendre pour que JVMTI permette d'appeler des fonctions distantes.

Chargement de classes dans la JVM distante

Le chargement de classes additionnelles vitales au débogueur doit généralement s'effectuer avant l'évaluation de sa fonction `main`, afin que le débogueur et ses extensions soient opérationnels avant le début du programme utilisateur.

Pour mettre en œuvre le chargement de classes additionnelles, Bugloo pose automatiquement un point d'arrêt sur le chargement de la classe principale du programme. Lorsque le programme démarre, le chargement de cette classe provoque ainsi la suspension de la machine virtuelle et l'envoie d'un événement notifiant le passage sur ce point d'arrêt. Le débogueur peut alors charger les classes dont il a besoin et indiquer à ses extension de faire de même. Enfin, il commande à la JVM de reprendre son exécution normale.

Le chargement de classe revient à faire un appel de fonction distant dans la machine déboguée : on peut se servir des `ClassLoader` du langage Java pour forcer le chargement

d'une classe arbitraire puis demander à JVMTI de forcer son initialisation.

Mécanisme d'appel de fonction

Lorsque l'exécution du débogué est suspendue, le débogueur peut utiliser JDI pour effectuer un appel de fonction dans la JVM du débogué. L'API JDI impose que l'appel de fonction soit synchrone. Pour ne pas bloquer le débogueur (et en particulier l'écoute des événements JVMTI durant un appel), le débogueur doit « asynchroniser » l'appel. Pour cela il se sert des travailleurs Fair Threads.

Dans Bugloo, les appels de fonctions sont toujours effectués depuis un travailleur. Juste avant l'appel, le travailleur se désynchronise pour ne pas bloquer l'ordonnancement des Fair Threads. Lorsque l'appel distant se termine, le travailleur se resynchronise, enregistre une tâche future dans la boucle d'événements (cf. 3.5.2) qui réordonnancera les Fair Threads, puis s'endort temporairement. Plus tard, le *thread* graphique ordonnancera les Fair Threads, ce qui aura pour effet de réveiller le travailleur. Ce dernier pourra renvoyer la valeur de retour de l'appel distant asynchronisé.

Le problème du thread d'exécution

Lorsque le débogueur veut exécuter une fonction dans la JVM distante, il doit indiquer le *thread* distant à employer. Seul un *thread* suspendu par le déclenchement d'un événement JVMTI est éligible pour effectuer l'appel de fonction. Cela complique l'implantation de certaines fonctionnalités. Par exemple, pour pouvoir suspendre le débogué lorsque l'utilisateur tape la combinaison de touches `CTRL+C`, le débogueur doit charger une classe additionnelle dans le débogué :

```
1 public class suspend extends Thread {
2     public static void break() {};
3
4     public void run() {
5         while (true)
6             try { wait(); } catch (Exception _) { break(); }
7     }
8 }
```

Le débogueur place automatiquement un point d'arrêt dans la fonction `break`. Lorsque l'utilisateur tape `CTRL+C`, le débogueur réveille le *thread* créé par la classe `suspend` en forçant une interruption. Ainsi, le *thread* entre dans `break`, l'exécution se suspend sur le point d'arrêt et le débogueur dispose d'un candidat valide pour exécuter des appels de fonction.

3.6 Performances du débogueur et pénalités à l'exécution

Il est important de vérifier que les performances du débogueur soient suffisamment bonnes pour déboguer des programmes de taille conséquence comme le compilateur Bigloo lui-même (envrion 130000 lignes).

Pour mesurer les pénalités qu'implique l'instrumentation des programmes JVM, nous avons exécuté des tests de performances sur quatre programmes écrits en Bigloo : Fib est un petit programme qui calcul la suite de Fibonnacci ; QSort est un programme conçu pour tester les tableaux Scheme et les calculs à virgule fixe ; Peval est un évaluateur partiel qui crée beaucoup de fermetures afin de tester l'allocateur mémoire ; Cgc est un compilateur de langage ressemblant à C qui produit du code assembleur MIPS.

Les tests ont été réalisés avec un Pentium 4 HT 3.4Ghz, 1024 Mo sous Linux 2.6.15-23 avec le JDK 1.6.0-rc-b71 de Sun (Hotspot). La charge de la machine était minimale. Pour chaque programme testé, l'utilisation mémoire et le temps d'exécution ont été mesurés pour une exécution normale et pour une exécution instrumentée par Bugloo. Les temps sont recueillis à l'aide de la commande `time`. Le temps user est conservé, car nous voulons uniquement mesurer les pénalités de compilation, et non les appels système ou les changements de contexte dûs aux multiples *threads* s'exécutant dans la JVM. À titre de comparaison, des tests similaires ont été réalisés sur les mêmes programmes compilés en code natif et instrumentés avec le débogueur GDB.

	(vide)	Fib	Qsort	Peval	Cgc
Jvm	_	7.01 s	29.79 s	10.56 s	20.54 s
Bugloo	$\mathcal{T}_0 = 2.32$ s	9.52 s	32.56 s	13.54 s	24.13 s
$\delta(\mathcal{T}_0)$	_	+2.76 %	+1.50 %	+6.23 %	+6.22 %
Native	_	8.69 s	16.67 s	8.69 s	6.59 s
GDB	$\mathcal{T}_1 \approx 0$ s	8.75 s	38.64 s	10.56 s	35.19 s
$\delta(\mathcal{T}_0)$	_	+0.69 %	+131.79 %	+21.52 %	+433.99 %

Fig. 3.8: Pénalités de l'exécution des programmes instrumentés par un débogueur

Les résultats des tests de performances sont exposés dans la figure 3.8. Les tests mémoire ne sont pas reportés car la consommation mémoire n'est pas affectée par l'exécution de la JVM en mode débogage. Le temps d'exécution des programmes est toujours légèrement plus lent avec Bugloo. Dans le test *vide*, $\mathcal{T}_0$ représente le temps que met la JVM du débogueur pour charger Bugloo puis terminer immédiatement. On peut donc considérer que $\frac{\text{Bugloo}-\mathcal{T}_0}{\text{Jvm}}$ est une approximation raisonnable de la pénalité occasionnée par le débogueur. Il en ressort que Bugloo entraîne un ralentissement compris entre 1.50 % et 6.23 %. Il semble donc clair que le JIT de la JVM — dont le but est de compiler à la volée du code octet JVM en code natif — maintient ses optimisation durant la compilation en mode débogage. En fait, lorsque l'utilisateur suspend l'exécution, la JVM remplace le code compilé optimisé présent en sommet de pile par sa version interprétée originale. Cette approche permet à l'utilisateur de déboguer correctement le programme lorsque l'exécution est suspendue, tout en conservant des performances optimales durant l'exécution. En revanche, il n'en va pas du tout de même pour la version C des programmes. En effet, pour pouvoir déboguer un programme correctement sur cette plateforme d'exécution, il ne faut plus utiliser certaines passes d'optimisation du compilateur afin de conserver les variables locales ou les appels de fonctions. Cela a des effets souvent catastrophiques comme le montre la chute de 433 % pour le test Cgc !

En conclusion, on voit que le JIT reste efficace durant l'instrumentation d'une JVM. Manifestement, les chutes de performances dépendent de la fonctionnalité de débogage utilisée. Par exemple, dans la JVM de Sun, l'exécution pas-à-pas est exécutée en mode interprété. Hormis cela, un gros programme comme le compilateur Bigloo peut aisément être débogué, ce qui indique que les chutes de performances restent modérées.

3.7 Travaux reliés

Bugloo n'est pas le seul débogueur à fournir un langage de commande pour *scripter* le débogué. Des débogueurs classiques comme DBX ou GDB ont un langage de commandes complet qui permet à des outils de débogage spécialisés de s'interfacer. C'est le cas du

débogueur Bigloo BDB, ou des outils de débogage de code CLI fournis avec la machine .Net de Mono[3]. Toutefois, l'interfaçage par la ligne de commande a plusieurs inconvénients. Tout d'abord, l'outil externe doit résider dans son propre processus et communiquer avec GDB uniquement par chaînes de caractères. Cela est moins pratique qu'une API de programmation et plus lent. De plus, le coût de maintenance est très élevé car il faut modifier le code du débogueur à chaque évolution du langage de commande de GDB.

DDD [ZL96] est une interface graphique aux débogueurs populaires comme GDB, JDB ou Perl Debugger. Son interface, proche de GDBX [Bas86], propose de visualiser sur un plan 2D des objets structurés et leurs pointeurs. Il propose en plus une notation pour décrire la manière dont doivent être affichées les structure simples ou emboîtées. Les objets sont automatiquement réorganisés quand l'un d'entre eux est expansé. BUGLOO ne fournit pas par défaut ce genre du vue. Toutefois, son système d'inspecteur structuré permettrait d'implanter ce type de visualisation.

Le débogage de certains langages peut poser des problèmes avec JVMTI. Par exemple, les implantations de langage Chez Scheme [Dyb98], Scheme 48 [KR94] ou MIT Scheme [Han91] intègrent un débogueur dans lequel la pile d'exécution contient les sous-expressions en attente d'évaluation. Pour JVMTI, la pile contient des fonctions JVM et la granularité des informations de débogage est la ligne. En revanche, en construisant des information de lignes spéciales (cf. chapitre 6), BUGLOO peut fonctionner avec une exécution « caractère par caractère » au lieu d'une exécution « ligne par ligne », comme cela est prévu par la JVM.

À notre connaissance, il n'existe pas de débogueur pouvant gérer correctement les programmes composés de plusieurs langages de haut niveau. Strein et Kratz on développé un débogueur [SK05] multi-langages pour .NET, mais il suppose que les langages aient une compilation directe. L'environnement de programmation Python JyDT utilise l'API de débogage fournie par l'éditeur Eclipse pour déboguer le langage de haut niveau Jython, mais il ne prend pas en compte le débogage multi-langages. BUGLOO semble être le seul débogueur à mettre en place des mécanismes extensibles pour prendre en compte la compilation délicate des langages de haut niveau.

3.8 Conclusion

Ce chapitre a présenté un aperçu de BUGLOO, un débogueur pour tout langage compilé vers la JVM. Ce débogueur peut être contrôlé à la ligne de commande ou s'utiliser dans l'environnement de programmation Emacs. Il utilise une bibliothèque graphique afin de permettre l'inspection des objets structurés de manière efficace.

Les fonctionnalités du débogueur sont toutes exportées dans une interface de programmation écrite en langage Scheme. Cette interface a un double usage : les utilisateurs s'en servent pour contrôler le débogueur à la ligne de commande et pour le personnaliser ; les implanteurs s'en servent pour construire des modules d'extension afin de supporter le débogage de leur langage.

BUGLOO est conçu pour déboguer efficacement les langages nécessitant une compilation complexe. Les programmes peuvent disposer de plusieurs strates d'informations de débogage. De plus, l'inspection des données repose sur des mécanismes génériques spécialisables : les implanteurs peuvent fixer le comportement de la recherche des variables. Ils

[3] voir http ://www.go-mono.com.

peuvent aussi construire une représentation virtuelle des objets structurés de leurs langages et personnaliser leur affichage.

Chapitre 4

Filtrage de la pile d'exécution

Le chapitre précédent a présenté une vue d'ensemble de Bugloo, notamment les mécanismes dont il dispose pour masquer l'apparition de structures et de variables synthétiques dûes à la compilation des langages de haut niveau. Ce chapitre décrit le deuxième type de traitement mis en œuvre pour masquer les détails de compilation de ces langages. Ces mécanismes permettent tout d'abord de construire une vue virtuelle de la pile d'exécution durant l'inspection, afin de masquer les fonctions produites par le compilateur ou bien encore de visualiser dans une seule pile différentes représentations de code (typiquement du code natif et du code-octet). Ces mécanismes permettent aussi d'effectuer une exécution pas-à-pas contrôlée, afin de ne jamais s'arrêter dans une fonction synthétique produite par le compilateur. La suite du chapitre présente les notations qui permettent de décrire le contenu de la pile d'exécution. Puis, sont décrits les mécanismes de construction de vue virtuelle et d'exécution contrôlée. Le chapitre se termine par une comparaison avec les travaux similaires dont le but est de masquer les détails de compilation qui apparaissent dans la pile d'exécution.

4.1 Le problème des blocs d'activation synthétiques

Comme évoqué dans le chapitre 1, la compilation de certaines constructions des langages de haut niveau (par exemple les fermetures) vers une plateforme d'exécution généraliste comme la JVM est une tâche complexe. Les transformations que subit un programme source durant sa compilation peuvent handicaper son débogage. Une des causes de cette difficulté de débogage est la présence dans la pile d'exécution de fonctions « privées » absentes du code source du programme. La figure 4.1 illustre les principaux problèmes de débogage que cela peut entraîner.

Le programme Java de la figure 4.1(b) donne une vision simplifiée du résultat de la compilation du programme Bigloo de la figure 4.1(a). Ce programme contient un nom d'identificateur encodé, ce qui n'est pas gênant lorsque le débogueur dispose de mécanismes de visualisation adaptés (cf. chapitre 3). En revanche, la présence de fonctions synthétiques entraîne deux problèmes majeurs. Tout d'abord, l'inspection de la pile de ce programme laisse apparaître ces artefacts de compilation, ce qui interfère avec l'image mentale que l'utilisateur a de l'exécution de son programme. De plus, ces artefacts perturbent fortement l'exécution pas-à-pas. Par exemple, un saut effectué pour sortir de la fonction `hello` amènera dans la fonction synthétique `funcall1`.

```
(module test
   (main go!))

(define (hello s)
   (print "hello " s))

(define (go! args)
   (let ((f hello))
      (f "world")))
```

(a) programme original

⇝

```
public class test extends proc {
    procedure ptr_hello;
    procedure ptr_BgL_goz12z12;

    public Objet funcall1(proc p, Object arg) {
        switch (p.id) {
        case 1: return hello(arg);
        case 2: return BgL_goz12z12(arg);
        default: return bbool.faux;
        }
    }

    public Object hello (Object s) {
        System.out.println("hello"+s);
    }

    public Object BgL_goz12z12(Object args) {
        procedure f=ptr_hello;
        return f.funcall1(args);
    }

    public static void main(String[] argv) {
        ptr_hello=new test(1);
        ptr_BgL_goz12z12=new test(2);
        BgL_goz12z12(runtime.wrap_args(argv));
    }
}
```

(b) programme compilé

FIG. 4.1: Exemple de pollution apparente après la compilation d'un programme

4.1.1 Nature des blocs d'activations synthétiques

La pollution de la pile d'exécution peut survenir lorsque des abstractions de niveau « langage » sont implantées dans une bibliothèque d'exécution. Par exemple, la bibliothèque de programmation GNOME [War04] fournit un système d'objet portable avec typage dynamique pour les programmes écrits en C pur. Les fonctionnalités comme la création d'objet ou la conversion de type dynamique (*casts*) sont implantées dans des fonctions qui apparaissent dans la pile d'exécution, ce qui n'est pas désirable pour les débogueurs.

Certaines abstractions des langages de haut niveau sont implantées en produisant des fonctions synthétiques intermédiaires. Par exemple, les appels récursifs terminaux peuvent être émulés avec des techniques dites de trampoline [Bak95, Cli98, SO01] qui utilisent une fonction de contrôle pour appeler des fonctions utilisateur. Quand une fonction veut faire un appel terminal, elle retourne la fonction à appeler à la fonction de contrôle, qui n'a plus qu'à faire un appel de fonction standard. Ce type d'approche est extrêmement embêtant pour les débogueurs car l'information utile dans la pile peut se retrouver *noyée* dans un ensemble d'appels de fonctions synthétiques.

Parfois, la sémantique du langage source diffère tellement de celle de la plateforme d'exécution qu'une compilation directe est impossible. Dans ce cas, la compilation produit du code s'appuyant sur un interprète *ad-hoc* qui gère sa propre pile à l'exécution. C'est le cas par exemple de Jython [Hug97], qui embarque son propre interprète dans la JVM. Dans ce cas, les débogueurs standards ne sont plus capables d'instrumenter le programme et deviennent donc complètement inutiles.

4.1.2 Solution pour expurger la pile d'exécution

Un débogueur doit fournir un moyen de masquer les détails de compilation précédemment évoqués. Il doit par là-même conserver son efficacité pour déboguer des programmes multi-langages. Pour y parvenir, Bugloo met à disposition des implanteurs de langages un nouveau mécanisme de construction de vue virtuelle de la pile qui repose sur quatre notions fondamentales. La première, appelée *filtre de bloc*, est une notation générique permettant d'identifier un bloc d'activation dans la pile d'exécution. La deuxième notion, appelée *motifs de blocs*, permet de construire des motifs réguliers de blocs d'activation afin de détecter des séquences connues de motifs de code produits par les compilateurs de langage de haut niveau. La troisième notion est un algorithme guidé par des règles capables de construire une *vue virtuelle* de la pile d'exécution, dans laquelle les blocs d'activation indésirables ont été supprimés et où les informations logiques qui avaient disparues de la pile après la phase de compilation peuvent être reconstruites. C'est grâce à cette notion que Bugloo peut afficher correctement la pile de programmes utilisant un ou plusieurs langages de haut niveau. Enfin, la dernière notion est un algorithme permettant d'effectuer une *exécution pas-à-pas contrôlée* pour éviter de s'arrêter dans des fonctions synthétiques produites par le compilateur. Elle permet aussi d'adapter l'effet d'un saut en fonction de la nature du code en cours d'exécution.

Les mécanismes développés pour le filtrage de la pile sont volontairement simples pour limiter leur coût et en même temps suffisamment puissants pour traiter complètement des langages de haut niveau. Les implanteurs de langage peuvent bénéficier de ces techniques de filtrage pour leur langage en développant un ensemble de règles adéquat.

La suite de ce chapitre présente la notion de filtre de bloc. Puis nous décrivons le langage de construction de motifs de blocs et présentons des exemples d'utilisation. Enfin, le reste du chapitre décrit successivement les deux mécanismes mis au point pour construire des vue virtuelles et pour effectuer une exécution pas-à-pas contrôlée.

4.2 Détection de bloc d'activation

Cette section présente la première notation développée pour permettre d'identifier des blocs d'activation contenus dans la pile d'exécution. Tout d'abord, une définition de la pile et des blocs d'activation est donnée. Puis, la section décrit la notion de *filtre de bloc*, qui permet de vérifier qu'une fonction représentée par un bloc d'activation possède certaines propriétés (comme par exemple un nom ou des types d'arguments particuliers).

4.2.1 Définitions d'un bloc d'activation

Soit τ un point donné dans le temps pendant l'exécution d'un programme P. La *pile d'exécution* S du programme P au temps τ est définie comme suit :

$$S_\tau(P) = f_1.f_2 \ldots f_n$$

où les f_i représentent les *blocs d'activation*, c'est-à-dire les appels de fonction en attente dans P au temps τ. Le bloc d'activation f_1 est le plus récent dans la pile : il représente le contexte d'exécution au temps τ. Dans notre modèle, un bloc d'activation est représenté par le 7-uplet suivant :

$$f \in \mathsf{Frame} = \langle name, args, ret, def, file, line, env \rangle$$

Les éléments de ce N-uplet sont appelés des *attributs* : *name* est une chaîne de caractères représentant le nom de la fonction ; *args* est une liste de chaînes de caractères représentant le type de chacun des arguments de *name* ; *ret* est une chaîne de caractères représentant le type de retour de *name* ; *def* est une chaîne de caractères représentant l'endroit où la fonction *name* est définie[1] ; *file* représente le chemin du fichier source dans lequel la fonction est définie ; *line* est un entier qui représente la position dans le code source à laquelle l'exécution s'est suspendue à l'instant τ. Enfin, *env* est un ensemble de triplets $\langle n, t, v \rangle$ représentant toutes les variables locales visibles (en comptant les arguments de la fonction) dans f à l'instant τ. Chaque triplet représente respectivement le nom d'une variable locale, son type et sa valeur.

4.2.2 Les filtres de bloc

La première étape vers la construction d'une représentation virtuelle de pile est de pouvoir identifier des blocs d'activation dans la pile. Pour cela, il est nécessaire de disposer d'une fonction de comparaison. Intuitivement, on serait tenté de définir un prédicat d'égalité entre deux blocs d'activation qui soit vrai si les attributs des blocs sont égaux deux à deux. Dans la pratique, un test d'égalité *stricte* est trop restrictif. En fait, un prédicat reposant sur des *motifs* est mieux adapté pour la localisation d'un bloc d'activation, car cela permet des comparaisons plus fines comme :

- comparer uniquement certains attributs d'un bloc d'activation. Par exemple, si l'on souhaite rechercher une fonction définie dans un fichier source particulier ;
- faire des comparaisons complexes. Par exemple lorsqu'on recherche une fonction dont le nom commence par un préfixe particulier.

La figure 4.2 présente l'algorithme de détection de bloc d'activation. Dans cet algorithme, nous introduisons une notation que nous appelons *filtre de bloc* qui permet d'identifier des classes de blocs d'activation :

$$\phi \in Filter = \langle name, args, ret, def, file \rangle$$

Un filtre de bloc est un 5-uplet identique à un bloc d'activation sans ses deux derniers attributs : ces derniers sont inutiles pour identifier un bloc d'activation car leur valeur varie selon l'endroit où l'on se trouve dans une fonction.

Pour détecter un bloc d'activation, on utilise la fonction match, qui effectue une comparaison entre un bloc et un filtre. Dans la figure 4.2, les a_i représentent les sept attributs d'un bloc d'activation et les α_i les cinq attributs d'un filtre. Pour qu'il y ait correspondance entre un filtre et un bloc, ses cinq attributs doivent deux à deux correspondre aux cinq premiers attributs d'un bloc. La comparaison de deux attributs (appelée m^i dans la figure) est effectuée par la fonction match_atom, à l'exception de a_2 et α_2 qui sont comparés à l'aide de la fonction match_list, car ce sont des listes d'attributs.

La fonction match_atom prend en paramètre l'attribut d'un bloc d'activation et l'attribut correspondant dans un filtre. Le type de l'attribut du bloc est toujours une chaîne de caractères, comme expliqué précédemment. En revanche, le type de l'attribut du filtre peut varier, comme indiqué dans la figure 4.2. Ce peut être une chaîne de caractères, une expression régulière, une fonction utilisateur ou une valeur spéciale dénotée `any`. Le test de comparaison employé dépend directement de ce type :

[1] Dans le cas de la JVM, *def* représente le nom d'une classe JVM. Dans un autre contexte, cela pourrait être le nom d'une bibliothèque partagée.

$$\mathsf{match} : \mathsf{Frame} \times \mathsf{Filter} \times \mathsf{Memory} \longrightarrow \mathsf{Bool} \times \mathsf{Memory}$$

$$\mathsf{match}(\langle a_1, \ldots, a_7\rangle, \langle \alpha_1, \ldots, \alpha_5\rangle, M) = \begin{cases} \langle \mathit{true}, M'_5\rangle & \text{si } m^i(a_i, \alpha_i, M_i) = \langle \mathit{true}, M'_i\rangle, \forall i \in [1,5]; \\ \langle \mathit{false}, \emptyset\rangle & \text{sinon} \end{cases}$$

$$\text{avec } m^x = \begin{cases} \mathsf{match\_list} & \text{si } x = 2; \\ \mathsf{match\_atom} & \text{sinon} \end{cases} \quad \text{et} \quad M_x = \begin{cases} M & \text{si } x = 1; \\ M'_{x-1} & \text{sinon} \end{cases}$$

$$\mathsf{match\_atom} : \mathsf{String} \times \mathsf{FilterAtom} \times \mathsf{Memory} \longrightarrow \mathsf{Bool} \times \mathsf{Memory}$$

$$\mathsf{match\_atom}(a, \alpha, M) = \begin{cases} \langle \mathsf{string\_equal}(a, \alpha), M\rangle & \text{si } \alpha \in \mathsf{String}; \\ \mathsf{regex\_match}(a, \alpha, M) & \text{si } \alpha \in \mathsf{Regex}; \\ \alpha(a, M) & \text{si } \alpha \in \mathsf{Function}; \\ \langle \mathit{true}, M\rangle & \text{si } \alpha = \texttt{any}; \\ \mathsf{match\_atom}(a, x, M.a) & \text{si } \alpha \equiv (\texttt{?}\ x); \\ \langle \mathsf{string\text{-}equal}(a, M[n]), M\rangle & \text{si } \alpha \equiv (\backslash\ n)\ \wedge\ 1 \leq n \leq |M| \end{cases}$$

$$\mathsf{match\_list} : [\mathsf{String}] \times [\mathsf{FilterAtom}] \times \mathsf{Memory} \longrightarrow \mathsf{Bool} \times \mathsf{Memory}$$

$$\mathsf{match\_list}(L_1, \texttt{any}, M) = \langle \mathit{true}, M\rangle \qquad \mathsf{match\_list}(L_1, \texttt{any}, M) = \langle \mathit{true}, M\rangle$$
$$\mathsf{match\_list}(\emptyset, \emptyset, M) = \langle \mathit{true}, M\rangle \qquad \mathsf{match\_list}(\emptyset, \emptyset, M) = \langle \mathit{true}, M\rangle$$

$$\mathsf{match\_list}(E, (\texttt{?}\ \mathrm{F}), M) = \mathsf{match\_list}(L_1, L_2, M.E)$$

$$\mathsf{match\_list}(E, (\backslash\ \mathrm{n}), M) = \begin{cases} \mathsf{match\_list}(E, M[n], M) & \text{si } 1 \leq n \leq |M|\ \wedge\ M[n] \notin \mathit{String}; \\ \langle \mathit{false}, M\rangle & \text{sinon} \end{cases}$$

$$\mathsf{match\_list}(a.L_1, \alpha.L_2, M) = \begin{cases} \langle \mathit{false}, \emptyset\rangle & \text{si } \mathsf{match\_atom}(a, \alpha, M) = \langle \mathit{false}, \emptyset\rangle; \\ \mathsf{match\_list}(L_1, L_2, M') & \text{si } \mathsf{match\_atom}(a, \alpha, M) = \langle \mathit{true}, M'\rangle \end{cases}$$

FIG. 4.2: L'algorithme de détection d'un bloc d'activation

String : match_atom compare cette chaîne de caractères à la valeur de l'attribut correspondant dans le bloc d'activation ;

Regex : match_atom cherche si l'attribut appartient au langage régulier dénoté par l'expression régulière ;

Function : match_atom applique directement cette fonction à la valeur de l'attribut correspondant dans le bloc d'activation. Les fonctions servent à effectuer des comparaisons arbitraires ;

`any` : c'est une valeur spéciale pour laquelle la fonction match_atom renvoie toujours *vrai*. Elle est utilisée pour indiquer que la valeur de l'attribut correspondant n'est pas importante.

Dans la suite, on suppose qu'il existe un moyen de discriminer le type des attributs d'un filtre passés à la fonction match_atom durant le filtrage.

La fonction match_list compare le second attribut d'un bloc d'activation avec l'attribut correspondant d'un filtre. Le premier argument de la fonction est une liste de chaînes de caractères, le second argument est quant à lui une liste d'objets pouvant prendre l'un des quatre types décrits précédemment. Cette fonction de comparaison renvoie *vrai* si les deux listes sont de tailles égales et si leurs éléments sont égaux deux à deux. Les éléments sont comparés entre eux à l'aide de la fonction match_atom.

Exemples de filtres de bloc

Dans l'implantation réalisée pour BUGLOO, un filtre est représenté par une liste Scheme de cinq éléments. Un attribut de type chaîne de caractères est représenté par une simple chaîne Scheme. Une expression régulière est représentée par une chaîne Scheme entourée par des crochets. Une fonction utilisateur est représentée par son nom sous forme de symbole Scheme. Enfin, la valeur spéciale `any` est représenté par le symbole Scheme équivalent. À titre d'exemple, voici la syntaxe concrète d'un filtre qui caractérise les fonctions appelées `foo`, qui prennent un seul argument de type `int` et qui retournent un `int` :

```
("foo" ("int") "int" any any)
```

On peut aussi produire des constructions plus complexes, comme par exemple le filtre suivant qui détecte les fonctions dont le nom commence par `display_` et qui prennent au moins deux arguments :

```
(["display_.*"] (any any . any) any any any)
```

Le point à l'intérieur de la liste a la même signification que la notation Lisp servant à décrire la structure des listes [McC60]. En effet, dans notre notation, une liste d'arguments est une liste Lisp : soit la liste vide, soit un doublet dont le premier élément est un argument et le second est une liste d'arguments. Dans notre exemple, le troisième symbole *any* peut donc détecter la liste vide (si la fonction à reconnaître a *exactement* deux arguments), ou une liste d'arguments (si la fonction a plus de deux arguments).

Enfin, on peut utiliser dans un filtre une fonction utilisateur annexe pour effectuer des comparaisons arbitrairement complexes. Par exemple, BUGLOO fournit une fonction utilitaire `get-type-descriptor` retournant un objet qui décrit un type utilisé dans le programme débogué. Il fournit aussi un prédicat `subtype?` qui détermine si un type hérite d'un autre. À l'aide de ces deux fonctions on peut construire un filtre qui détecte les fonctions définies dans des classes JVM qui héritent de la classe `java.lang.Listener` :

```
(any any listener-test any any)
```

en supposant qu'il existe une fonction utilisateur `listener-test` qui prend un nom de classe en paramètre :

```
(define (listener-test class)
   (subtype? (get-type-descriptor class) "java.lang.Listener"))
```

4.2.3 Support des références dans les filtres

L'utilisation de simples expressions régulières est insuffisante pour détecter des répétitions, comme par exemple le fait que deux attributs d'un bloc d'activation doivent avoir la même valeur. Pour obtenir ce niveau d'expressivité, il est nécessaire de disposer d'une

fonctionnalité similaire aux *références* des expressions régulières POSIX [IEE93]. Cela se traduit par l'ajout de deux formes composées dans l'algorithme :

1. lorsqu'un attribut de bloc est de la forme `(? a)`, la valeur de l'attribut correspondant dans le bloc d'activation est *mémorisée*, puis on compare cette valeur avec l'attribut de bloc `a` ;
2. lorsqu'un attribut de bloc est de la forme `(\ n)`, on compare la valeur de l'attribut correspondant dans le bloc d'activation avec la `n`-ième valeur mémorisée.

Dans l'algorithme présenté en figure 4.2, les fonctions de détection M partagent un argument mémoire, servant à sauvegarder la valeur des attributs capturés durant le test de détection. Ces valeurs peuvent être utilisées pour effectuer une sorte de « détection conditionnelle ». La mémoire est une liste ordonnée contenant les valeurs des attributs. Durant la détection, chaque nouvelle valeur capturée est ajoutée en queue de mémoire. En fin de test, la fonction m^i retourne un doublet qui contient un booléen indiquant s'il y a eu correspondance entre les deux attributs testés et l'état de la mémoire M'_i après exécution de m^i. On voit dans la figure que les attributs de blocs d'activation et de filtres de blocs sont comparés entre eux de manière séquentielle. L'état de la mémoire est successivement passé d'un test de détection à un autre, comme indiqué dans la définition de M_x.

Exemple d'utilisation des références

À l'aide des références, il est possible d'écrire un filtre permettant de détecter des fonctions dont les deux premiers arguments doivent être de même type, quelque soit la valeur de ce type :

```
(any ((? any) (\ 1) . any) any any any)
```

Il est possible d'utiliser des expressions régulières *étendues* : ce type d'expression régulière peut capturer des portions de la chaîne de caractères à détecter en utilisant les méta-caractères « `(` » et « `)` » dans son motif. Comme pour les attributs de filtres, lorsqu'une séquence de caractères est capturée, elle est ajoutée en queue de mémoire. De manière symétrique, la `n`-ième valeur capturée peut être référencée dans le motif en utilisant la méta-caractère `\` suivie du nombre `n`. La figure 4.2 montre que la fonction regex_match accepte une mémoire en paramètre et retourne un couple ⟨booléen, mémoire⟩. On peut remarquer que la mémoire provient de la fonction match_atom et qu'il est donc possible de référencer dans une expression régulière une valeur capturée par une autre expression régulière ou même par des attributs de type `(? a)`. Pour illustrer cette capacité, voici un filtre détectant les fonctions nommées `print_`α et ayant un paramètre de type α, quelle que soit la valeur de α :

```
(["print_(.*)"] ((\ 1)) any any any)
```

Lorsque la détection commence, la mémoire est initialement vide. Quand le nom de la fonction à comparer correspond à l'expression régulière, sa tête est capturée et ajoutée en queue de mémoire. Puis la détection compare le nom du type du paramètre de la fonction avec l'attribut de filtre `(\ 1)`, c'est-à-dire la valeur capturée dans l'expression régulière.

La mémoire M contient généralement des chaînes de caractères, mais elle peut aussi contenir des listes de chaînes de caractères au cas où la mémorisation ait eu lieu dans match_list. Si la fonction string_equal reçoit une liste de chaînes de caractères, elle retourne proprement la valeur *faux*. En fait, capturer une liste d'attributs présente un intérêt

lorsqu'on veut comparer le type des arguments de deux fonctions différentes présentes dans la pile. Ce cas de figure sera décrit en détail dans la section suivante.

4.3 Détection de motifs de blocs d'activation dans la pile

Grâce aux filtres de blocs, il est possible d'identifier un bloc d'activation arbitraire dans la pile. Pour pouvoir décrire la configuration de la pile, il faut maintenant un moyen de détecter si un groupe de blocs d'activation consécutifs se trouve sur la pile. Cela nécessite l'introduction d'un nouveau langage de motif nommé *Omega*. Ce langage permet de décrire des successions de blocs d'activation dans la pile. Dans ce nouveau type de détection de motif, analogue aux expressions régulières classiques, la chaîne de caractères est remplacée par la pile d'exécution et les caractères par les blocs d'activation.

4.3.1 Le langage de motif Omega

Omega est composé de caractères qui sont en fait des filtres de bloc et d'un ensemble de méta-caractères permettant la construction et la composition de motifs de bloc d'activation. À la différence des expressions régulières classiques, les mots du langage *Omega* sont représentés sous une forme « pré-digérée » qui reflète la structure logique arborescente d'un motif. Les mots du langage sont construits comme indiqué ci-dessous :

$$\omega \in \mathit{Omega} = \begin{cases} \epsilon & \text{mot vide;} \\ \phi \in \mathsf{Filter} & \text{filtre de bloc;} \\ \omega_1.\omega_2 & \text{concaténation de deux mots de } \mathit{Omega}\text{;} \\ (\neq\ \phi_1 \cdots \phi_n) & \text{complément de l'ensemble de filtres de bloc } \phi_i\text{;} \\ (\mid \omega_1 \cdots \omega_n) & \text{choix de l'un des mots } \omega_i\text{;} \\ (* \ \omega) & \text{répétition du mot } \omega \text{ (étoile de Kleene [Kle56])}\text{;} \\ (? \ \omega) & \text{apparition optionnelle de } \omega\text{, équivalent à } (\mid \omega\ \epsilon)\text{;} \\ (+ \ \omega) & \text{au moins une répétition de } \omega\text{, équivalent à } \omega.(* \ \omega) \end{cases}$$

La détection de motif de blocs s'effectue grâce à la fonction match_stack, dont l'algorithme est présenté dans la figure 4.3. Cette fonction de détection prend en paramètre une séquence de blocs d'activation qui sont consécutifs dans la pile d'exécution du débogué, un motif de pile $\omega \in \mathit{Omega}$ et une mémoire initialement vide servant à la capture d'attributs de blocs d'activation. En retour, elle renvoie un triplet dont le premier élément est un booléen indiquant s'il existe un entier $i \leq n$ tel que les i premiers blocs d'activation de la séquence soient reconnus par ω. En cas de détection, le second argument représente les i premiers blocs et le troisième argument, les blocs d'activation restant dans la liste.

La détection d'une suite consécutive de blocs d'activation est un algorithme itératif similaire à la détection de motif dans une chaîne de caractères. À chaque étape, les arguments de la fonction match_stack ont la forme suivante :

$$\mathsf{match\_stack}(f_1 \ldots \uparrow f_i \ldots f_n, \omega.\Omega, M)$$

Dans la figure, un pointeur dénoté par une flèche verticale apparaît dans le premier argument pour indiquer la progression de la détection. À gauche du pointeur se trouvent les blocs

$$\mathsf{match\_stack} : [\mathsf{Frame}] \times \mathsf{Omega} \times \mathsf{Memory} \longrightarrow \mathsf{Bool} \times [\mathsf{Frame}] \times [\mathsf{Frame}]$$

fonction $\mathsf{match\_stack}(f_1 \ldots \uparrow f_i \ldots f_n, \omega.\Omega, M)$:
si $\omega.\Omega \equiv \emptyset$ alors
$\mathsf{match\_stack} \leftarrow \langle true, f_1 \ldots f_i, f_{i+1} \ldots f_n \rangle$
sinonsi $\omega \equiv \phi$ et $\mathsf{match}(f_i, \phi, M) = \langle true, M' \rangle$ alors
$\mathsf{match\_stack} \leftarrow \mathsf{match\_stack}(f_1 \ldots f_i \uparrow f_{i+1} \ldots f_n, \Omega, M')$
sinonsi $\omega \equiv (\neq \phi_1 \ldots \phi_m)$ et $\forall j \in [1, m] \mid \mathsf{match}(f_i, \phi_j, M) = \langle false, M' \rangle$ alors
$\mathsf{match\_stack} \leftarrow \mathsf{match\_stack}(f_1 \ldots f_i \uparrow f_{i+1} \ldots f_n, \Omega, M')$
sinonsi $\omega \equiv (\mid \omega_1 \ldots \omega_m)$ et $\exists j \in [1, m] \mid \mathsf{match\_stack}(\uparrow f_i \ldots f_n, \omega_j.\Omega, M) = \langle true, F, G \rangle$ alors
$\mathsf{match\_stack} \leftarrow \langle true, f_1 \ldots f_{i-1}.F, G \rangle$
sinonsi $\omega \equiv (*\omega')$ et $\mathsf{match\_stack}(\uparrow f_i \ldots f_n, \omega'.(*\omega').\Omega, M) = \langle true, F, G \rangle$ alors
$\mathsf{match\_stack} \leftarrow \langle true, f_1 \ldots f_{i-1}.F, G \rangle$
sinonsi $\omega \equiv (*\omega')$ et $\mathsf{match\_stack}(\uparrow f_i \ldots f_n, \omega'.(*\omega').\Omega, M) = \langle false, \emptyset, \emptyset \rangle$ alors
$\mathsf{match\_stack} \leftarrow \mathsf{match\_stack}(f_1 \ldots \uparrow f_i \ldots f_n, \Omega, M)$
sinon
$\mathsf{match\_stack} \leftarrow \langle false, \emptyset, \emptyset \rangle$
finsi

FIG. 4.3: L'algorithme de détection de motif de blocs.

qui ont déjà été reconnus par le motif de pile initial Ω_0 ; à droite du pointeur se trouvent les blocs restant dans le flot d'entrée. Le second argument $\omega.\Omega$ représente ce qu'il reste du motif initial Ω_0 après avoir détecté $i-1$ blocs dans le flot d'entrée.

Lorsque la détection commence, le pointeur se trouve à l'extrémité gauche du flot de blocs d'activation. Si un groupe de blocs se trouvant directement à droite du pointeur correspond au motif ω, le pointeur avance d'un bloc d'activation dans la séquence et le motif à détecter est réduit à Ω. Ainsi, une séquence de blocs d'activation est reconnue par un motif de bloc si ce dernier peut être réduit à $\emptyset$ par une suite d'itérations de l'algorithme. En corollaire, on peut noter que les blocs du flot ne doivent pas forcément être consommés en totalité pour que la correspondance ait lieu.

4.3.2 Exemples de syntaxe concrète

La syntaxe concrète des motifs de blocs se base sur l'utilisation de listes Lisp. Les caractères du langage *Omega* sont des filtres de bloc. La concaténation de motifs est n-aire et elle est représentée par une liste Lisp dont les éléments sont les motifs. Les autres méta-opérations sont représentées par un doublet dont le premier élément est le méta-caractère et le second une liste d'opérandes (des filtres de bloc ou des motifs de blocs comme vu précédemment). Pour illustrer cette syntaxe, voici un motif de blocs permettant de détecter les séquences de blocs d'activation de la forme |`foo`, `bar`, ...| ou |`foo`, `gee`, `hux`, ...| et qui rejette les séquence comme |`hux`, `bar`, `foo`, ...| :

```
[("foo" any any any any)
 (| ("bar" any any any any)
    ("gee" any any any any))]
```

Dans un motif de pile, on peut utiliser les références aux valeurs capturées pour permettre une comparaison structurelle entre plusieurs blocs d'activation. La détection de motif se sert de la fonction match pour tester la correspondance entre un filtre du motif et un bloc

d'activation dans la pile. En sortie de fonction, l'état de la mémoire est conservé pour être ensuite passé en paramètre au prochain appel à match. De cette manière, on peut détecter des motifs de manière « contextuelle ». Par exemple, on peut écrire un motif qui détecte dans la pile une fonction de nom et d'argument arbitraire, immédiatement suivie d'une fonction du même nom préfixé par un tiret et ayant la même configuration d'argument[2] :

```
[((? "foo") (? any) any any any)
 ("_\\1" (\ 2) any any any)]
```

4.3.3 Les motifs de pile comme condition d'arrêt

Le langage *Omega* offre un moyen de décrire le contenu de la pile d'exécution et permet aux implanteurs de langage de définir des motifs de blocs d'activation à masquer ou à transformer afin de pouvoir présenter à l'utilisateur une pile expurgée des artefacts de compilation. Mais ce langage se révèle aussi très intéressant pour obtenir des points d'arrêt conditionnels activés lorsque la pile d'exécution se trouve dans une configuration particulière. Par exemple, il devient possible de suspendre l'exécution dans la fonction `foo` d'un programme `prog1` seulement si `foo` a été appelée depuis la fonction `bar` :

```
(bp add prog1 foo :stack ((foo any any any any)
                          (bar any any any any)))
```

Dans Bugloo, on utilise l'attribut `:stack` pour activer un point d'arrêt si le motif associé correspond au sommet de la pile d'exécution.

4.4 Construction d'une vue virtuelle de la pile d'exécution

La section précédente a décrit la capacité du débogueur à détecter des motifs de blocs d'activation contenus dans la pile d'exécution. Cette section introduit la notion de *vue virtuelle de pile*, qui consiste à construire une représentation de la pile d'exécution dans laquelle les informations non désirables résultant de la compilation des langages de haut niveau sont cachées. La méthode développée permet de visualiser différentes représentations de code disjointes (comme du code natif et du code interprété) dans une seule et unique pile virtuelle. Cette section présente tout d'abord les principes de la vue virtuelle. Puis, elle décrit l'algorithme permettant sa construction et un exemple d'utilisation.

4.4.1 Principes de la construction de la vue virtuelle

La construction d'une vue virtuelle de pile s'effectue en extrayant de l'information sémantique de la pile d'exécution brute : on utilise un motif de pile pour localiser un groupe de blocs d'activation correspondant à un motif de code connu produit par un compilateur. Ce groupe est remplacé dans la vue virtuelle par un ou plusieurs blocs représentant uniquement des fonctions « logiques », c'est-à-dire des fonctions définies par l'utilisateur dans le code source. En itérant ce processus sur l'ensemble de la pile brute, on obtient une vue virtuelle de la pile originale où tous les détails non désirables issus de la compilation ont été masqués et où l'information perdue lors de la compilation a pu être reconstruite. Selon sa nature, un motif de bloc d'activation peut être :

[2]Le double anti-slash dans le second filtre provient de la notation C pour écrire un anti-slash dans une chaîne de caractères.

– caché dans la vue virtuelle, au cas où les blocs d'activation représentent uniquement des appels de fonctions intermédiaires produits par le compilateur ;
– remplacé par un ou plusieurs blocs d'activation détectés par ce motif. Par exemple, quand un appel de fonction utilisateur est précédé d'un appel de fonction engendré par le compilateur, seul l'appel utilisateur est conservé dans la vue virtuelle ;
– remplacé par un ou plusieurs « blocs virtuels » qui n'existent pas dans la pile originale. Par exemple, si le groupe de blocs d'activation détecté représente l'interprète du langage, il doit être remplacé par un bloc virtuel qui représente la véritable fonction utilisateur en train d'être évaluée par l'interprète.

La construction de la vue virtuelle de pile est une technique générique. Ainsi, pour pouvoir masquer les détails de compilation d'un langage particulier, les implanteurs du langage ont seulement besoin de fournir un ensemble de *règles de substitution*, une règle étant composée d'un motif de pile chargé de détecter un motif de code synthétique et d'une action à effectuer pour remplacer les blocs d'activation détectés.

4.4.2 L'algorithme de construction de la vue virtuelle

$$\mathsf{virtual}_T : [\mathsf{Frame}] \times [\mathsf{Rule}] \times \mathsf{Subst} \longrightarrow [\mathsf{Frame}]$$

$$\mathsf{Subst} : [\mathsf{Frame}] \longrightarrow [\mathsf{Frame}] \longrightarrow [\mathsf{Frame}]$$

$$\frac{}{\mathsf{virtual}_T(\emptyset, R, S) = \emptyset}$$

$$\frac{\mathsf{match\_stack}(f_1 \ldots f_n, \omega, \emptyset) = \langle \mathit{false}, F, F' \rangle}{\mathsf{virtual}_T(f_1 \ldots f_n, \langle \pi, \omega, \lambda_\rho, \lambda_\sigma \rangle . R, S) = \mathsf{virtual}_T(f_1 \ldots f_n, R, S)}$$

$$\frac{}{\mathsf{virtual}_T(f_1 \ldots f_n, \emptyset, S) = \mathsf{pending}(S, f_1).\mathsf{virtual}_T(f_2 \ldots f_n, T, \emptyset)}$$

$$\frac{\mathsf{match\_stack}(\langle f_1 \ldots f_n, \omega, \emptyset \rangle) = \langle \mathit{true}, f_1 \ldots f_i, f_{i+1} \ldots f_n \rangle \quad \lambda_\rho(f_1 \ldots f_i) = g_1 \ldots g_j}{\mathsf{virtual}_T(f_1 \ldots f_n, \langle \pi, \omega, \lambda_\rho, \lambda_\sigma \rangle . R, S) = \mathsf{pending}(S, g_1 \ldots g_j).\mathsf{virtual}_T(f_{i+1} \ldots f_n, R, \emptyset)}$$

$$\frac{\mathsf{match\_stack}(\langle f_1 \ldots f_n, \omega, \emptyset \rangle) = \langle \mathit{true}, f_1 \ldots f_i, f_{i+1} \ldots f_n \rangle \quad \lambda_\rho(f_1 \ldots f_i) = \emptyset}{\mathsf{virtual}_T(f_1 \ldots f_n, \langle \pi, \omega, \lambda_\rho, \lambda_\sigma \rangle . R, S) = \mathsf{virtual}_T(f_{i+1} \ldots f_n, R, \lambda_\sigma(f_1 \ldots f_i).S)}$$

$$\text{avec } \mathsf{pending}(S, f_1 \ldots f_n) = \begin{cases} f_1 \ldots f_n & \text{si } S = \emptyset; \\ \mathsf{pending}(S', \sigma(f_1 \ldots f_n)) & \text{si } S \equiv \sigma.S' \end{cases}$$

$$\text{et } \sigma : [\mathsf{Frame}] \longrightarrow [\mathsf{Frame}]$$

FIG. 4.4: l'algorithme de construction de la vue virtuelle

La figure 4.4 présente l'algorithme complet de construction de vue virtuelle. La fonction $\mathsf{virtual}_T$ est la transformation qui retourne une vue virtuelle de la pile pour un ensemble donné de *règles de remplacement* T. L'algorithme est découpé en cinq cas qui représentent les différentes itérations possibles durant la construction. La partie au dessus de la barre horizontale représente les conditions devant être réalisées pour qu'une transition soit choisie pour l'itération suivante. La partie au dessous de la barre représente le résultat de la prochaine itération. Une règle de remplacement est un quadruplet de la forme :

$$r \in \mathsf{Rule} = \langle \pi, \omega, \lambda_\rho, \lambda_\sigma \rangle$$

L'élément π est un entier qui représente la priorité de la règle de remplacement r dans l'ensemble T, au cas où plusieurs règles seraient applicables à un endroit donné de la pile brute. Dans ce cas, la règle choisie pour le remplacement serait celle qui a l'entier le plus petit. La valeur π est arbitraire : elle est fixée par l'implanteur du langage en fonction de la complexité des règles qu'il a définies.

L'élément ω est un motif de pile, il définit la portion de pile que la règle r peut remplacer. L'élément λ_ρ est une fonction que l'on applique à l'ensemble des blocs d'activation détectés par ω pour obtenir l'ensemble des blocs d'activation qui resteront dans la vue virtuelle.

La construction de la vue virtuelle est un processus itératif qui commence au sommet de la pile et qui se déplace vers les blocs d'activation plus profonds. À chaque itération, les blocs accessibles depuis la position courante sont comparés à chaque motif ω définis dans T pour savoir s'il est possible d'appliquer un remplacement. Si c'est le cas, les blocs détectés par cette règle sont remplacés et la construction de la vue continue avec les blocs restants dans la pile. Durant la transformation, l'algorithme ne fait pas d'itération sur les blocs d'activation qui résultent d'un remplacement : la construction se poursuit à partir des blocs d'activation restant dans la pile brute. Si aucune règle n'est applicable pour une position donnée dans la pile brute, le bloc d'activation courant est considéré *propre*, il est conservé tel quel dans la vue virtuelle et la construction continue un bloc plus bas dans la pile.

L'élément λ_σ est appelé une *substitution retardée*. C'est un argument optionnel employé lorsque la séquence de bloc détectée par la règle ne correspond pas à un bloc d'activation entier dans la vue virtuelle, mais seulement à un ou plusieurs attributs de blocs (typiquement l'information de ligne ou les variables locales). Dans ce cas, ces attributs sont « mémorisés » jusqu'à ce qu'un nouveau bloc d'activation soit ajouté à la vue virtuelle, puis ces attributs sont reportés dans ce bloc. L'algorithme de la figure 4.4 indique que la λ_σ est une fonction prenant en paramètre la séquence de blocs d'activation détectée par ω et retournant une fermeture de type σ : [Frame]$\longrightarrow$[Frame] qui a mémorisé les attributs devant être conservés. Chaque fois qu'un remplacement r est appliqué, sa substitution retardée est ajoutée à la liste des substitutions en attente dénotée S. Dès qu'un nouveau bloc est ajouté dans la vue virtuelle, la fonction pending applique toutes les substitutions en attente au bloc d'activation.

4.4.3 Exemple de construction de vue virtuelle

L'exemple de la figure 4.5 illustre l'utilisation des différents types de remplacement. À gauche de la figure se trouve la pile brute d'un programme qui a commencé son exécution dans une fonction `main`, qui a ensuite fait un appel d'ordre supérieur à la fonction `foo` et qui a atteint un point d'arrêt dans cette fonction à la ligne 7. Supposons qu'à cet endroit du programme, l'exécution se trouve dans un bloc de traitement d'exception. Dans cette pile, les fonctions n'ayant pas d'information de ligne représentent des fonctions synthétiques produites par le compilateur pour implanter le traitement des exceptions et les appels d'ordre supérieur. La compilation du bloc de traitement d'exception a produit la fonction synthétique `__try_foo1`.

Supposons que dans le but de masquer ces deux motifs de code, l'implanteur du langage a déjà conçu une règle de remplacement r_0 qui détecte les blocs d'activation 3 et 4 pour masquer l'implantation des appels d'ordre supérieur. Il doit maintenant concevoir une règle r_1 telle que les trois premiers blocs d'activation par un bloc virtuel représentant la fonction `foo` à la ligne 7. La règle ne doit pas rechercher le troisième bloc d'activation dans la

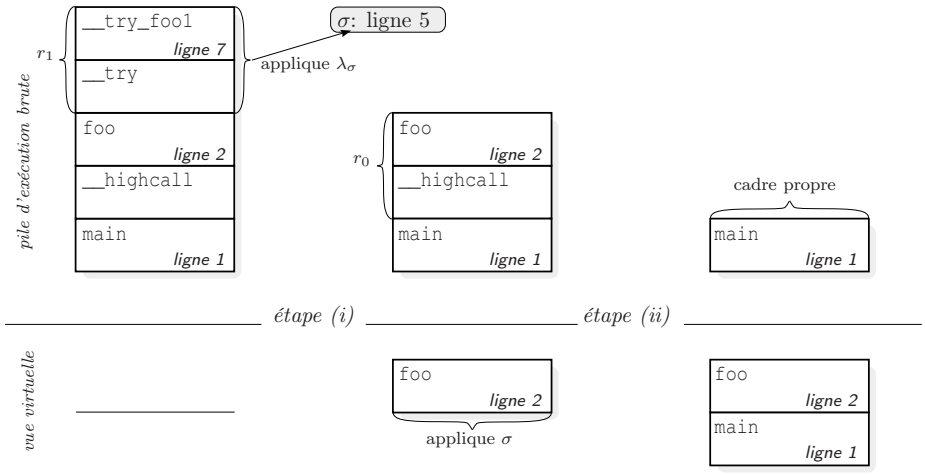


FIG. 4.5: exemple de substitution retardée

pile car cela empêcherait la règle r_0 de s'appliquer[3]. Il est donc nécessaire de procéder en plusieurs étapes. Tout d'abord, l'implanteur conçoit r_1 pour que seulement les deux premiers blocs d'activation soient détectés et masqués et pour créer une substitution retardée σ qui mémorise l'information « ligne 7 ». Cela ne crée pas de bloc dans la vue virtuelle et termine l'étape *(i)* dans le schéma. Ensuite la règle r_0 remplace les blocs d'activation 3 et 4 par une fonction `foo` localisée à la ligne 2, ce qui nous donne la vue virtuelle temporaire au milieu du schéma. À ce moment, la substitution σ peut être appliquée au bloc de la vue virtuelle pour corriger son information de ligne. Cela termine l'étape *(ii)*. Enfin le dernier bloc d'activation dans la pile brute n'a pas besoin d'être transformé et apparaît donc tel quel dans la vue virtuelle, ce qui termine sa construction.

4.4.4 Syntaxe concrète

Dans la syntaxe concrète, une règle de remplacement est représentée par une liste de trois à quatre éléments, selon que l'on se serve de λ_σ ou pas. À titre d'exemple, voici comment on peut écrire une règle de priorité 10 masquant les occurrences de la fonction `foo` dans la pile si elles sont précédées de la fonction `bar` ou de la fonction `gee` :

```
(10
 [("foo" any any any any)
  (| ("bar" any any any any)
     ("gee" any any any any))]
 mask)
```

La fonction `mask` est définie par `(λ (frames) '())`. Cette règle ne contient pas de substitution retardée. Le chapitre suivant présentera des exemples concrets d'utilisation des règles de remplacement, ainsi que des utilisations des substitutions retardées.

[3] car un bloc d'activation dans la pile brute ne peut être détecté qu'un seule fois durant la construction de la vue.

4.5 Contrôle de l'exécution pas-à-pas

Le débogueur Bugloo permet d'effectuer trois types de sauts durant l'exécution pas-à-pas :

- le saut *entrant* avance d'un pas dans l'exécution. Si la prochaine instruction dans le code source est un appel de fonction, le saut emmène à l'intérieur de cette fonction. Sinon, il fait avancer d'une ligne dans le code source ;
- le saut *de ligne* avance jusqu'à la prochaine ligne dans le code source de manière inconditionnelle. Tous les appels se trouvant avant la prochaine ligne sont exécutés durant le saut ;
- le saut *sortant* permet de continuer l'exécution jusqu'à sortir de la fonction courante.

Comme expliqué au début de ce chapitre, la compilation des langage de haut niveau introduit souvent dans le code utilisateur des appels intermédiaires à des fonctions de bibliothèque ou des fonctions synthétiques qui nuisent au débogage durant l'exécution pas-à-pas. Par exemple, certaines implantations de langages manipulent les nombres entiers à travers des « boîtes » allouées en tas [JL91]. L'allocation de ces boîtes entraîne des appels à leurs constructeurs, ce qui perturbe l'exécution pas-à-pas du code utilisateur.

Pour éviter les arrêt superflus, les débogueurs traditionnels permettent de définir un ensemble de fichiers ou de bibliothèques dans lesquels l'exécution pas-à-pas ne doit pas s'arrêter[4]. Or, pour tenir compte du fait que la compilation des langages de haut niveau ajoute des fonctions synthétiques au niveau du code utilisateur, il est nécessaire d'adapter le mécanisme de filtrage de saut traditionnel :

- il faut permettre de filtrer les sauts « par fonction » et non plus par fichier ou par bibliothèque, afin d'isoler les fonctions synthétiques intermédiaires qui sont produites par la compilation d'un module de code utilisateur ;
- pour s'assurer qu'une fonction représente bien du code intermédiaire, il faut pouvoir consulter la pile afin de s'assurer que les fonctions en sommet de pile représentent bien un motif de code synthétique connu. Pour cela, on peut réutiliser le langage *Omega* présenté dans la section 4.3.1 ;
- il faut mettre au point un mécanisme de contrôle spécifique pour éviter les sauts superflus. Par exemple, lorsqu'un saut provoque la sortie d'une fonction utilisateur et le retour dans une fonction synthétique, il faut pouvoir continuer l'exécution jusqu'à revenir dans une fonction utilisateur ;
- il faut pouvoir adapter le comportement de l'exécution pas-à-pas en fonction du contexte d'exécution, afin de fournir une exécution pas-à-pas correcte même en présence d'évaluation de fonctions interprétées.

La suite de la section décrit les mécanismes mis en œuvre dans le débogueur pour contrôler l'exécution pas-à-pas en fonction des motifs de code présents en sommet de pile.

4.5.1 Mécanismes de contrôle de l'exécution

Contrôle de l'exécution pas-à-pas

Dans le but de masquer les sauts indésirables causés par la compilation complexe des langages, les implanteurs de langage doivent définir un ensemble de filtres, nommés *sauts virtuels*, qui spécifient les configurations de pile dans lesquelles l'exécution ne doit pas

[4] Dans JVMTI, la granularité des sites est la classe ou le package JVM.

s'arrêter et les actions à effectuer pour « continuer » l'exécution jusqu'à revenir dans du code utilisateur. Un saut virtuel s est représenté par un triplet de la forme :

$$s \in \mathsf{Step} = \langle \pi, \omega, \alpha \rangle$$

L'élément ω est un motif de bloc qui permet de déterminer dans quel contexte d'exécution le saut virtuel s peut être appliqué. En d'autres termes, s est applicable si la séquence de fonctions décrite par ω est présente en sommet de la pile brute.

L'élément α précise l'action à effectuer lorsque le saut s est appliqué. Deux types d'actions sont possibles :

`NEXT` : Cette action force l'exécution à continuer jusqu'à entrer dans un nouveau bloc d'activation ou, à défaut, jusqu'à sortir du bloc d'activation courant. On peut l'utiliser pour filtrer les appels de fonctions synthétiques intermédiaires dont le but est d'appeler des fonctions utilisateur ;

`OUT` : Cette action force l'exécution à sortir du bloc d'activation courant. On peut l'utiliser lorsque la fonction synthétique à filtrer fait uniquement appel à des fonctions synthétiques ou à des fonctions de bibliothèque.

Lorsqu'un saut virtuel entraîne la sortie d'un bloc d'activation, il se peut que l'exécution revienne dans une fonction synthétique devant être filtrée (par exemple, si un saut sort d'une fonction utilisateur). Dans ce cas, le débogueur émet automatiquement un autre saut similaire jusqu'à revenir dans un bloc d'activation utilisateur.

Un saut peut avoir deux types d'action et il est possible que plusieurs sauts soient applicables pour une configuration de pile donnée. C'est pour cela que chaque saut virtuel contient un entier π représentant sa priorité. Le saut choisi parmi ceux applicables est celui qui a l'entier le plus petit.

Exécution pas-à-pas contextuelle

Lorsque la construction de la pile virtuelle a eu lieu, le bloc d'activation en sommet de pile indique le contexte d'exécution « logique » tel que l'utilisateur le perçoit.

Dans le débogueur, les implanteurs de langage peuvent spécialiser le comportement des sauts pas-à-pas (entrant, de ligne et sortant) pour chaque type de bloc d'activation virtuel produit par une règle de remplacement. Cela permet de fournir une exécution pas-à-pas cohérente quelle que soit la nature du code masqué. Par exemple, lorsqu'un motif de code contient de nombreux bloc d'activations, l'implanteur du langage peut redéfinir le comportement du saut sortant afin de sortir de tous les blocs d'activation d'un coup. De manière similaire, si le sommet de pile contient une fonction interprétée utilisant du code-octet *ad-hoc*, l'implanteur du langage peut redéfinir le comportement des sauts pour qu'ils influent sur l'évaluation de ce code-octet.

4.5.2 Syntaxe concrète et exemple d'utilisation

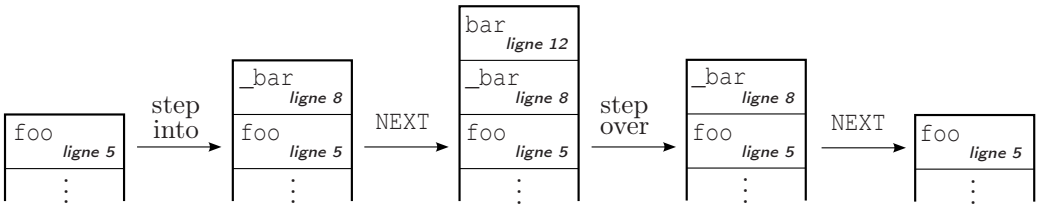
Tout comme une règle de remplacement, un saut virtuel est représenté par une liste Lisp à trois éléments : la priorité est un entier, le motif en langage *Omega* utilise la syntaxe présentée en section 4.3.2 et l'action à effectuer est représentée par les symboles Lisp `OUT` ou `NEXT`.

À titre d'exemple, la figure 4.6(a) présente un programme engendré contenant une fonction synthétique appelée `_bar` qui ne doit pas influencer l'exécution pas-à-pas. Supposons

```
1  (module stepping
2     (main foo))
3
4  (define (foo args)
5     (_bar 10)
6     (print "end foo"))
```

```
7  (define (_bar x)
8     (if (int? x) (bar x)
9         (print "error")))
10
11 (define (bar x)
12    (print "bar: " x))
```

(a) Le programme à exécuter pas-à-pas



(b) Filtrer `bar` avec la règle `NEXT`

FIG. 4.6: Filtrage d'un saut entrant

que l'exécution soit suspendue dans la fonction `foo` et qu'un saut entrant l'amène dans la fonction `_bar`. Le saut virtuel suivant permet d'éviter de s'arrêter dans cette fonction :

```
(bugloo) (step virtual add 100
                        [("__try" any any (? any) any)
                         (any any any (\ 1) any)] NEXT)
```

Dans cet exemple, le saut virtuel a une priorité 100. Son motif doit détecter la présence de `_bar` en sommet de pile. Pour s'assurer que la fonction est appelée dans le bon contexte, le motif vérifie qu'elle soit définie dans le même module que la fonction appelante (dans cas `foo`). Si le sommet de pile correspond, l'action `NEXT` prolongera le saut entrant pour arriver dans `bar`. Plus tard, si un saut sortant fait revenir l'exécution dans `_bar`, le saut virtuel sera réactivé et ramènera automatiquement l'exécution dans `foo`.

Dans l'exemple, le saut virtuel s'applique à l'entrée et à la sortie de fonction. Si la fonction `_bar` avait fait des appels à d'autres fonctions synthétiques après l'appel à `bar`, l'application du saut virtuel aurait engendré beaucoup de sauts implicites pour revenir jusqu'à `foo`, ce qui n'est pas très efficace. Si l'on suppose qu'un bloc d'activation virtuel remplace les fonctions `*bar` dans la vue virtuelle, il est possible de redéfinir le comportement du saut sortant pour faire exécuter à la JVM du débogué deux sauts sortants d'affilé afin de retourner automatiquement dans `foo`.

4.6 Implantation dans Bugloo

4.6.1 Implantation de l'inspecteur de pile d'exécution

Avec l'API de débogage JVMTI, les *threads*, les blocs d'activation et les variables locales sont réifiées : l'API fournit des objets pour inspecter l'état du débogué depuis la JVM du débogueur de manière transparente. BUGLOO enrobe ces objets dans ses propres objets pour masquer les détails de bas niveau et pour fournir des services étendus aux implanteurs de langages et aux utilisateurs du débogueur.

L'objet `thread`, accessible depuis le débogueur, permet de récupérer la pile d'exécution JVM d'un thread du débogué. Cette pile est représentée dans le débogueur par une liste d'objets `stackframe` qui correspondent aux blocs d'activation.

L'objet `stackframe` permet d'obtenir des informations concernant une fonction en attente dans la pile. On peut retrouver à partir de cet objet la valeur de tous les attributs d'un bloc d'activation (comme présentés dans la section 4.2.1), ainsi qu'un certain nombre d'autres informations :

- Les attributs représentant des types (les arguments de la fonction, son type de retour et le type de la classe JVM dans laquelle elle est définie) peuvent être retournés sous forme réifiée : l'information de type est alors contenue dans un objet `type` à partir duquel on peut rapidement obtenir des informations supplémentaires comme la chaîne d'héritage du type.
- L'objet `stackframe` peut retourner la position du bloc d'activation qu'il représente dans la pile d'exécution. Ce champ est utile pour un bloc virtuel afin de se rappeler de la position des vrais blocs qu'il remplace. Cela permet par exemple de consulter les variables locales d'un vrai bloc d'activation au lieu de celles présentées par le bloc virtuel.
- L'objet `stackframe` peut choisir un *demangler* préféré qui sera à même de gérer correctement son affichage (par exemple en notation préfixe pour les langages comme Lisp) ainsi que celui de ses variables locales. Cela évite au débogueur d'inférer l'information et apporte donc un gain de temps.
- L'objet `stackframe` peut fournir sa propre implantation des requêtes de saut en fournissant une liste d'association Scheme contenant les types de saut qu'elle surcharge, comme par exemple `'((:into #<proc>) (:over #<proc>) (:out #<proc>))`.

Les implanteurs de langages peuvent surcharger la classe `stackframe` pour représenter leur propres types de blocs d'activation. Par exemple, la surcharge permet de fournir des blocs d'activations représentant des fonctions interprétées *ad-hoc* qui n'utilisent pas le code-octet JVM. Dans ce cas, les accesseurs de la classe `stackframe` sont surchargés pour rechercher les valeurs des attributs du bloc d'activation directement dans la fonction interprétée.

Les triplets utilisés dans la sémantique pour représenter les variables locales d'un bloc d'activation JVM (cf. section 4.2.1) sont implantés par une classe `localvar`. On peut demander à cet objet le nom de la variable, la valeur que contient cette variable ou le type de cette valeur (ce qui renvoie un objet `type`). Le nom est correctement décodé à l'aide des mécanismes présentés dans le chapitre 3.

Les blocs virtuels peuvent définir des variables virtuelles, c'est-à-dire des objets qui ne sont pas des variables locales JVM. Par exemple, un bloc virtuel représentant l'interprète d'un langage gère ses propres variables locales. Ce type de variable est là encore obtenu en surchargeant la classe `localvar`.

Dans BUGLOO, tous les objets sont construits de manière *paresseuse*, pour éviter de faire transiter de l'information entre la JVM du débogué et celle du débogueur avant que l'utilisateur n'en ait réellement besoin. Par exemple, la pile d'exécution est une liste Scheme construite de manière paresseuse pour allouer uniquement les objets JVMTI des blocs d'activation que l'on inspecte. De même les objets JVMTI représentant les variables locales d'un bloc d'activation ne sont alloués qu'à la demande.

4.6.2 Performances de la construction de la vue virtuelle

Comme le reste des objets BUGLOO, la vue virtuelle de la pile d'exécution est construite de manière paresseuse. Cela est primordial pour certaines fonctionnalités du débogueur. Par exemple, chaque saut effectué durant l'exécution pas-à-pas provoque l'affichage du contexte d'exécution courant dans la vue virtuelle, ce qui nécessite uniquement de déterminer le premier bloc d'activation de cette vue. De même, les points d'arrêt conditionnels utilisant l'état de la pile virtuelle (cf. section 4.3.3) ont seulement besoin de connaître les premiers blocs de la vue. La construction paresseuse permet, dans ce cas, d'éviter une perte d'interactivité ou des dégradations de performances intolérables.

Dans notre implantation, le coût en terme de performance d'exécution de la construction de la vue virtuelle réside dans la fonction match_stack. En effet, une expression du langage *Omega* n'est pas compilée en un automate fini [Aho90] et ce pour deux raisons :

- l'alphabet du langage (c'est-à-dire les filtres de blocs) n'est pas fini, il autorise les expressions régulières et les appels aux fonctions utilisateur ;
- le langage *Omega* lui même n'est pas régulier [Aho90] du fait que l'on utilise des *références* et des méta-opérateurs comme le groupe ou le complément d'ensemble (cf. section 4.3.1).

À défaut de compilation, la simulation d'un mot de *Omega* est réalisée à l'aide d'un simple algorithme interprétatif avec retour arrière (ou *backtracking*). Cette approche facilite la mise en œuvre, mais elle a aussi une contrepartie : les répétitions dans le motifs sont traitées à l'aide du *backtracking*, ce qui rend les performances du simulateur proches de celles d'un évaluateur trivial de NFA [Ang79, Lar98], c'est-à-dire des performances de l'ordre de $O(2^n)$ dans les cas dégénérés.

Le fait que l'alphabet du langage ne soit pas fini entraîne la perte d'une autre optimisation. Durant la simulation d'un motif contenant des choix ou des répétitions, il n'est pas possible de déterminer l'action à réaliser en regardant simplement la valeur d'un bloc d'activation dans la pile. On doit prendre le problème dans le sens opposé : pour savoir comment continuer la simulation, il faut se servir du *backtracking* et essayer toutes les actions.

Les performances mitigées de notre implantation ne sont toutefois que théoriques. En pratique, les performances de la construction de la vue virtuelle dépendent directement du nombre et de la complexité des règles de remplacements associées au langage. Dans la pratique, les motifs *Omega* mis en œuvre sont toujours d'une complexité raisonnable. Même si certains tests sont dupliqués dans différents motifs, un langage entier comme Bigloo (cf. chapitre 6) peut être pris en charge sans problème de performance sur notre machine de test (un Pentium M 1.4 GHz).

4.6.3 Support du filtrage de l'exécution pas-à-pas

Dans BUGLOO, il est possible d'utiliser le filtrage traditionnel ainsi que les sauts virtuels. Ces deux types de filtrage ont leurs avantages et leurs inconvénients, ce qui les rend complémentaires.

Le filtrage traditionnel est fourni en natif par l'API JVMTI. Il est accessible dans le débogueur par la commande suivante :

```
(bugloo) (filter class add '("*.Impl" "com.sun.*"))
```

Les chaînes de caractères représentent les endroits dans lesquels on ne souhaite pas s'arrêter. Ces chaînes sont des expressions régulières « bridées », qui supportent l'utilisation d'un seul méta caractère *Joker* et cela uniquement en tête ou en queue de la chaîne.

Lorsqu'un saut est effectué, ces filtres sont directement traités par la JVM du débogué. Lorsqu'un saut est filtré, la JVM du débogué n'envoie pas de notification « saut effectué » au débogueur. Ce dernier n'intervient donc aucunement dans le traitement des filtres, ce qui évite de faire transiter de l'information entre les deux JVMs. Le coût de ce filtrage est donc quasi nul.

Les saut virtuels sont une fonctionnalité propre à BUGLOO et sont traités dans la JVM du débogueur. Il est donc nécessaire de faire transiter toutes les informations de saut entre les deux JVMs, plus les commandes que le débogueur renvoie au débogué pour mettre en œuvre ces sauts.

Le type de requêtes envoyées vers la JVM du débogué pour reprendre temporairement son exécution dépend de l'action à déclencher :

- pour un saut ayant une action `OUT`, le débogueur requiert l'exécution du saut sortant. Lorsque ce dernier a abouti, le débogueur affiche à l'écran le nouveau contexte d'exécution ;
- pour un saut ayant une action `NEXT`, le débogueur pose deux points d'arrêt temporaires pour détecter la prochaine entrée ou sortie de fonction. Puis il demande à la JVM du débogué de reprendre l'exécution. Dès que l'une des deux actions a provoqué la suspension du débogué, le débogueur détruit les requêtes temporaires et affiche à l'écran le nouveau contexte d'exécution, en faisant croire que c'est un saut (et non pas un point d'arrêt) qui est à l'origine du message.

Le traitement des sauts dans le débogueur et les communications nécessaires entre les deux JVMs font que les sauts virtuels sont plus lents que le filtrage natif. De plus, il peut arriver que le résultat d'un saut virtuel amène l'exécution dans un point qui nécessite un autre saut virtuel. Le coût exact d'un saut virtuel ne peut donc pas être connu à l'avance. Dans la pratique, cette différence de performance entre les deux types de saut est négligeable, car l'utilisateur lui-même marque souvent un temps de réflexion après l'exécution d'un saut.

4.7 Travaux reliés

4.7.1 environnement de débogage pour langages de haut niveau

Les compilateurs de langages de haut niveau visant les plates-formes d'exécution natives ne fournissent généralement pas de support de débogage pour les programmes qu'ils produisent. BDB [Ser00] est un débogueur conçu pour déboguer les programmes Bigloo compilés en code natif, qui utilise GDB pour instrumenter le processus débogué. Il peut afficher indifféremment les fonctions C ou Bigloo présentes dans la pile, mais ne peut pas afficher ou déboguer correctement l'interprète du langage. Certains systèmes sont plus évolués, comme par exemple VisualWorks [How95]. Cet interprète de Smalltalk peut être lancé dans un débogueur classique pour inspecter la pile native. De plus, il exporte des fonctions à appeler depuis le débogueur pour inspecter la pile du programme Smalltalk. Le débogage d'un tel système n'est pas idéal car chaque type de code (natif et Smalltalk) possède ses propres fonctions d'inspection et de contrôle de l'exécution. De plus, l'implantation de la VM de VisualWorks est exposée dans la pile native. Enfin, on ne peux pas visualiser les deux types de code dans une vue virtuelle unique.

La capacité de déboguer efficacement les programmes de haut niveau compilés vers la JVM varie selon les implantations. Par exemple, l'interprète Jython [Hug97] inclut les

blocs d'activation JVM et les blocs d'activation Python dans ses traces de pile d'exécution. En revanche, les programmes Jython doivent être débogués avec un outil spécifique conçu pour Python, ce qui interdit le débogage d'autres langages. De même, Rhino [Fou98] compile directement du code ECMAScript [ECM99] en code-octet JVM, mais les arguments des appels de fonctions produits par le compilateur sont cachés à l'intérieur de structures de données intermédiaires. De plus, il introduits trop d'appels de fonctions intermédiaires pour être débogable efficacement avec un débogueur standard comme JSWAT [Fie99]. AspectJ [KHH+01] offre un bon support de débogage. Par exemple, le compilateur produit différentes strates d'informations de manière à permettre un débogage correct des aspects qui ont été tissés dans du code utilisateur. Par contre, l'encodage du nom des aspects présents dans la pile d'exécution n'est pas correctement décodé durant le débogage. La vue virtuelle de Bugloo aurait pu masquer ce genre de détail.

Certaines approches de débogage sont conçues pour tenir compte de l'aspect « multi-langage » des programmes utilisateurs. Par exemple, le débogueur `mdb` [Sun02] supporte le débogage de code natif dans l'esprit de GDB, mais fournit en plus un mode spécial pour le code JVM. Dans une session de débogage, on peut changer de vue pour visualiser la pile JVM, la pile JNI [5] ou la pile native. Toutefois il est impossible d'obtenir une vue unique de ces trois mondes.

Le système d'information de débogage en *strates* [Fie03], apparu dans le JDK 1.4, est une avancée intéressante pour les outils de transformation de code source, comme par exemple les compilateurs JSP vers Java. Ce système suppose qu'une fonction peut avoir plusieurs « vues » (dans notre cas JSP ou Java) et permet de fournir des informations de lignes différentes pour chaque vue. On peut ainsi masquer les sauts indésirables résultant de la transformation de code. En revanche, il est impossible de masquer les appels de fonctions intermédiaires. De plus, cette vue en strates se limite à la fonction et non à la pile comme c'est le cas avec le concept de pile virtuelle.

La machine virtuelle .NET a été conçue dans un soucis d'inter-opérabilité des langages. Ainsi, le débogueur intégré dans l'environnement Microsoft Visual Studio supporte le débogages des programmes composés de plusieurs langages et utilise les bibliothèques de chaque langage pour désencoder le nom des objets et afficher leur valeurs. De plus, il permet de visualiser une pile qui contient à la fois des fonctions écrites en code-octet CLI [Lid02] et des fonctions natives. Actuellement, Bugloo n'est pas capable de présenter les fonctions natives dans ses vues virtuelles car la JVM ne fournit pas le support d'inspection de pile nécessaire. En revanche, le débogueur .NET ne fournit pas de moyen de masquer les abstractions de langage implantés dans des fonctions de bibliothèques comme les fonction génériques de Bigloo. Le concept de bloc d'activation virtuel est un moyen simple et générique d'y parvenir.

4.7.2 Origine des techniques d'abstraction de pile

L'idée de construire une vue virtuelle de la pile d'exécution est apparue en même temps que les premiers compilateurs de code optimisant. Depuis, beaucoup de travaux ont été réalisés pour que les débogueurs ne soient pas ou peu affectés par les optimisations du flot de contrôle d'une programme. Zellweger appelle cela l'*expected behavior* et le *truthfull behavior* [Zel83, Zel84] et propose une méthode pour construire des blocs d'activation logiques pour pouvoir visualiser dans la pile les appels de fonctions ayant disparu à cause

[5] les fonctions natives définies dans des classes Java

de l'*inlining*. DOC [CMR88] est un compilateur produisant des informations de débogage spéciales afin de présenter des informations de lignes correctes et de retrouver les valeurs de variables qui ont disparu à cause de certaines optimisations (ordonnancement d'instructions, propagation de constantes, élimination de copies...). Dans leur travaux sur le débogage du langage Self [US87], les auteurs vont plus loin avec leur désoptimisation dynamique [HCU92]. Ils construisent des tables appelées descripteurs de portée [CUL89] qui permettent de retrouver, pour n'importe quel point d'une fonction optimisée, le ou les blocs d'activations logiques avant optimisation. Ainsi, lorsqu'on veut débogueur une fonction compilée, celle-ci est dépilée puis remplacée par la ou les fonctions équivalentes non optimisées. Cela permet d'effectuer toutes les opérations de débogage traditionnelles, y compris l'exécution pas-à-pas. Ces travaux ont directement été repris dans les JITs modernes comme le HotSpot de Sun ou le JIT d'IBM [SOT+00], où une fonction compilée avec optimisation (par exemple l'*inlining*) est remplacée par sa version non optimisée au moment du débogage. Cela semble être aussi le cas pour les JVMs de prochaine génération comme Jikes RVM avec son optimisation adaptative [AFG+00].

Les règles de remplacement ont été conçues pour recréer une vision logique d'un code source à partir de sa représentation compilée, c'est-à-dire pour remplacer un groupe de blocs d'activation dans la pile par un autre dans la vue virtuelle. Les techniques précédentes sont quant à elles capables de remplacer une fonction optimisée par plusieurs fonctions logiques. On ne peut pas obtenir ce résultat si l'on utilise uniquement nos techniques et des informations de débogage standard. Toutefois, notre approche fournit un moyen simple et générique de localiser des portions de pile, puis de les soumettre à un traitement utilisateur arbitraire. À ce titre, un implanteur de langage peut utiliser une règle de remplacement pour détecter une fonction optimisée contenant du code *inliné* et se servir de ces informations de débogage *ad-hoc* pour la remplacer par une suite de fonctions logiques dans la vue virtuelle. En conclusion, nous considérons que les techniques précédentes et notre algorithme de construction de vue virtuelle sont deux outils complémentaires.

4.8 Conclusion

Ce chapitre a présenté une méthode permettant de masquer les détails indésirables qui apparaissent dans la pile d'exécution et qui sont causés par la compilation complexe des langages de haut niveau vers des plates-formes d'exécution généralistes. Cette méthode simple et générique a été conçue pour permettre aux implanteurs de langages de haut niveau d'obtenir rapidement et à moindre coût un support de débogage complet pour leur langage.

La méthode développée s'articule autour de quatre notions fondamentales. La première notion, appelée filtre de bloc, permet d'inspecter les caractéristique d'un bloc d'activation. La seconde notion est un langage appelé *Omega* utilisant les filtres de bloc pour identifier des séquences de blocs d'activation. La troisième, appelée saut virtuel, se sert du langage *Omega* pour filtrer les sauts de l'exécution pas-à-pas, afin de ne pas s'arrêter dans des fonctions intermédiaires qui ne représentant que des détails de compilation. De plus, ce langage *Omega* a pu être réutilisé pour obtenir des points d'arrêt conditionnels activés selon l'état du sommet de pile. Enfin, la quatrième notion est une nouvelle technique de filtrage qui permet de remplacer des séquences de blocs d'activation par des blocs virtuels, c'est-à-dire des blocs qui ne sont pas physiquement présents dans la pile. En itérant ce processus sur toute la pile, on obtient une pile virtuelle dans laquelle chaque détail de compilation (appel de fonction ou structure de donnée intermédiaire) a disparu.

Les chapitres 6 et 7 présentent une utilisation des techniques de construction de pile virtuelle pour masquer les détails de compilation de différentes implantation de langages de programmation, ainsi que pour permettre le débogage de programmes utilisant plusieurs langages de haut niveau à la fois.

Chapitre 5

Débogage de l'allocation mémoire

La machine virtuelle Java gère automatiquement la mémoire au moyen d'un *Garbage Collector* (ou GC). Durant une session de débogage, Bugloo peut instrumenter l'allocateur mémoire, afin d'aider à localiser les bogues ou les problèmes de performances causés par des allocations trop fréquentes ou des rétentions mémoire. Ce chapitre présente les outils développés dans Bugloo pour le débogage mémoire, ainsi que les aménagements conçus pour les utiliser sur des langages de haut niveau comme Scheme. Le début du chapitre décrit le principe de fonctionnement des GCs et les types de bogues que ce genre d'allocateur peut entraîner. La suite du chapitre présente les différents outils de débogage mémoire développés dans ces travaux. Le premier, un inspecteur d'allocations, permet d'obtenir des informations sur l'occupation mémoire du tas et d'inspecter les sites d'allocation des objets. Le second, l'inspecteur de références, permet de visualiser graphiquement les différents types de chaînages entre les objets présents dans la mémoire et de comprendre la cause de certain bogues mémoires. Le dernier, un outil de profilage d'allocation, permettant de construire des statistiques exactes sur les allocations des programmes afin de détecter les points critiques et de résoudre des problèmes de performances dûs à des allocations abusives. La fin du chapitre compare les outils présentés aux travaux existants.

5.1 Plateformes d'exécution utilisant des GCs

5.1.1 Rappel sur le fonctionnement d'un GC

Dans les langages à désallocation explicite de mémoire tels que C, il peut se produire différents types de bogues mémoire pouvant entraîner une terminaison précoce de l'exécution :

- le programme tente de libérer une zone mémoire déjà libérée. Cela est une faute directe et entraîne un arrêt du programme ;
- le programme tente d'écrire dans une zone mémoire préalablement libérée. Si l'allocateur réutilise cette zone mémoire suite à une nouvelle allocation, cela peut provoquer des écrasements mémoire ;
- le programme détruit le dernier pointeur qui référençait une zone mémoire précédemment allouée, ce qui entraîne l'impossibilité de libérer cette zone par la suite. On appelle ce type de bogue une *fuite mémoire*. Son effet est de fragmenter la mémoire disponible et de provoquer des erreurs de type « mémoire insuffisante ».

Le principe d'un GC est d'éliminer ces types de problèmes en supprimant la désallocation manuelle : la mémoire est automatiquement libérée lorsqu'elle n'est plus utilisée par le programme.

La machine virtuelle Java ne fournit pas de type primitif « pointeur ». Les zones mémoires allouées dans le tas contiennent uniquement des objets JVM. Lorsque le programme a besoin de mémoire et que le tas est plein, le GC tente de libérer des zones mémoires occupées par des objets qui ne sont plus accessibles. On appelle cela un *ramassage*. Ce ramassage consiste à déterminer tous les objets que le GC peut atteindre à partir de pointeurs spéciaux appelés *racines*, comme par exemple un champ statique de classe, une variable locale dans la pile ou une référence native JNI. On dit de ces objets qu'ils sont *vivants*, car ils sont référencés par des racines ou des objets qui sont eux même vivants. Après avoir déterminé les objets encore vivants, le GC libère la place mémoire occupée par les autres objets du tas qui ne sont plus accessibles. On dit de ces objets qu'ils sont *morts*.

5.1.2 Débogage mémoire pour les architectures à GC

Les outils de débogage mémoire peuvent servir à comprendre les causes d'erreurs fatales dues à l'impossibilité d'allouer suffisamment de mémoire. Ces outils peuvent aussi servir à profiler un programme afin d'en exhiber les points qui consomment le plus de mémoire. Décrivons ci-dessous les solutions qu'apportent ces deux types d'outils.

Détecter la présence de fuites mémoires

Même lorsque l'on utilise des GCs, il peut arriver que les programmes se terminent anormalement si la quantité de mémoire libérée après un ramassage n'est pas suffisante pour le programme. C'est souvent le signe de la présence de fuite mémoire. En fait, même si les GCs permettent d'éviter les fuites mémoires des programmes C, ils sont toujours vulnérables à d'autres types de fuites :

- les objets qui sont censés être morts du point de vue du programmeur ne sont pas ramassés par le GC. Ce problème apparaît lorsque ces objets sont encore accessibles par le GC car ils sont encore référencés inutilement par le programme ;
- un objet servant de *zone de cache* (par exemple une table de hachage) grossit beaucoup plus vite que prévu. Cela se produit lorsque ce cache n'est pas « vidé » suffisamment fréquemment.

Le travail d'un débogueur est d'aider le programmeur à comprendre la cause de ces fuites en répondant à plusieurs questions fondamentales :

Quels sont les objets vivants ? Le débogueur peut interroger l'allocateur mémoire de la plateforme d'exécution afin d'obtenir le nombre d'objets présents dans le tas ainsi que leur type et leur taille. Cela permet de détecter des surconsommations mémoire potentielles. Cela permet aussi de détecter des fuites mémoires si des objets d'un certain type sont présents dans le tas de manière inattendue.

Pourquoi un objet est-il encore vivant ? Un objet est encore vivant s'il est accessible par une succession de pointeurs en partant d'une ou plusieurs racines du GC. Le débogueur doit donc pouvoir parcourir toutes les références liant les objets du tas pour retrouver la ou les racines responsables de la fuite mémoire.

Qui est responsable de l'allocation d'un objet ? Connaître la fonction responsable de cette allocation permet de réduire les recherches de la cause d'une fuite mémoire.

Pour permettre ce genre de requête de débogage, l'exécution doit être instrumentée de manière à conserver cette information durant toute la durée de vie d'un objet.

Profiler l'allocateur

Une mauvaise utilisation de la mémoire (c'est-à-dire des allocations trop nombreuses) peut entraîner des dégradations de performances à l'exécution. Il est donc utile de disposer d'un outil de « surveillance » des allocations effectuées par le programme. Ce type d'outil consiste en général à tracer les allocations se produisant durant l'exécution, puis à traiter les informations recueillis afin de répondre à différentes questions : combien d'objets a alloué une fonction donnée ? Quelles sont les fonctions qui ont alloué un type d'objet donné ? Combien d'objets ont été alloués entre chaque ramassage effectué par l'allocateur ?

5.1.3 Organisation de ce chapitre

La suite du chapitre présente les outils de débogage développés dans Bugloo. Ces outils mettent en œuvre les idées présentées précédemment en les adaptant pour prendre en compte les spécificités des langages de haut niveau : la première série d'outils permet d'inspecter les objets structurés du tas en prenant en compte les objets qui n'utilisent pas le système de type natif de la JVM ; le deuxième outil est un profileur d'allocation qui produit des statistiques qui prennent en compte les fonctions intermédiaires engendrées durant la compilation des langages de haut niveau.

5.2 L'inspecteur d'allocations

Cette section présente le premier outil développé dans Bugloo qui permet d'obtenir des statistiques sur le remplissage du tas et d'examiner les sites d'allocations des objets vivants. Dans la suite, nous décrivons son utilisation et son application aux langages de haut niveau.

5.2.1 Principe de fonctionnement

Lorsque l'exécution est suspendue, l'utilisateur peut demander des informations sur la taille et le taux de remplissage du tas :

```
(bugloo) (info heap)
16609 instances in 242 classes (total 1565592 bytes)
958432 bytes used in heap (total 14950400 bytes)
stats took 0.151s
```

La figure 5.1 présente la commande `(info heap)` dont le résultat est affiché dans l'environnement Emacs. L'inspecteur d'allocations présente une description détaillée de la consommation mémoire courante. Il permet d'obtenir la liste des objets (regroupés par type JVM) présents dans le tas, ainsi que la taille mémoire en octets qu'ils occupent. L'« instantané » ainsi obtenu peut être trié par nom de classe, par nombre d'objets ou par occupation mémoire. La fonction `(info heap)` accepte un paramètre optionnel permettant d'isoler les types JVM présentant un intérêt pour l'utilisateur à l'aide d'une expression régulière. Ainsi, par exemple, il est possible d'afficher uniquement les types Scheme (préfixés par `::`) à l'aide des paramètres suivants :

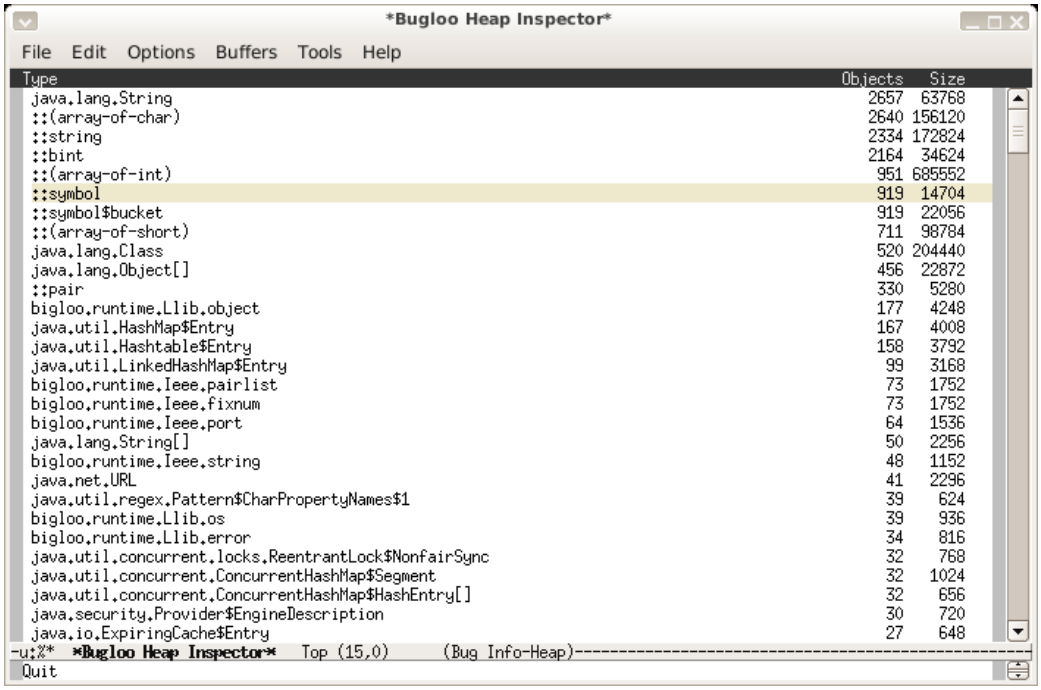


FIG. 5.1: Exemple de statistique de l'inspecteur d'allocation

```
(bugloo) (info heap "(::|bigloo)")
```

5.2.2 Inspecter les objets de types non-natifs

Les langages de haut niveau sont en général pourvus de leurs propres types d'objets structurés. Il peut arriver qu'il soit impossible de les représenter directement à l'aide du système de type natif de la JVM lorsque les spécificités des deux systèmes sont trop éloignées, voire incompatibles entre elles. Dans ce cas, la bibliothèque du langage met en œuvre son propre système de type et utilise des structures JVM intermédiaires pour représenter les objets utilisateurs. Grâce au mécanisme d'objets *virtuels* (cf. chapitre 3, section 3.4.2), ces objets sont affichés de manière transparente dans le débogueur.

Le débogueur peut produire des statistiques sur la quantité de mémoire occupée par les objets virtuels. Pour des raisons de simplicité, le choix a été fait de dissocier l'inspection des objets virtuels de celle des objets réels. Cela évite de décompter dans les statistiques les objets réels servant uniquement à implanter un objet virtuel.

Afin de comprendre l'utilité de l'inspection des types non-natifs, considérons le type « structure » du langage Bigloo. Durant l'évaluation d'un programme, la bibliothèque du langage permet de créer des structures bien qu'elle soit incapable de créer de nouvelles classes JVM à la volée. Ainsi, un objet structuré Bigloo de type `t` contenant deux champs `a` et `b` sera implanté par un objet JVM intermédiaire de type `bigloo.struct` contenant deux champs : le premier est un symbole encodant le nom `t`, le second est un tableau de taille deux contenant les valeurs des champs utilisateurs `a` et `b`. Durant une session de débogage, La commande suivante permettra d'afficher les objets virtuels du langage Bigloo :

```
(bugloo) (info heap :filter "::")
```

L'utilisation du mot-clé `:filter` indique au débogueur qu'il doit construire des statistiques sur les objets virtuels présents dans le tas. Pour cela, le débogueur produit tout d'abord des statistiques « classiques » sur les objets JVM. Ces statistiques sont ensuite traitées par les extensions de débogage actives dans le débogueur. L'extension en charge du langage Bigloo parcourra la liste des objets JVM de type `bigloo.struct` afin de déterminer la valeur des symboles présents dans les champs `t`. Cela lui permet de reconstruire la liste des types Bigloo « utilisateur ».

5.2.3 Inspecter l'allocation d'une partie du programme

Il est possible d'utiliser l'inspecteur d'allocation pour étudier la consommation mémoire d'une partie précise du programme dans laquelle on soupçonne un problème. Cette étude ciblée permet à l'utilisateur d'obtenir deux types d'informations :

- elle permet de mesurer de manière détaillée l'occupation mémoire engendrée par l'exécution d'une partie précise du programme. Par exemple, cela peut donner à l'utilisateur une indication sur l'efficacité de l'implantation d'un de ses algorithmes afin de confirmer ou d'infirmer ses intuitions ;
- elle permet de détecter la présence de fuites mémoire causées par un partie du programme. Pour cela, il suffit que l'utilisateur force le GC à ramasser les objets morts dans le tas puis qu'il demande des statistiques à l'inspecteur d'allocation. Les objets retournés sont vivants du point de vue du GC car ils ont résisté à la collection, mais en réalité ils sont « algorithmiquement » morts puisque la fonction étudiée a terminé son exécution. Cela est le symptôme d'une fuite mémoire dans la fonction étudiée.

Pour retrouver la quantité de mémoire dans le tas qui provient de l'exécution d'une partie du programme, le moyen le plus simple consiste à connaître le moment d'allocation de chaque objet. Il suffit alors de recherche tous les objets ayant été alloués durant l'exécution de la partie en question.

Bugloo instrumente l'allocateur mémoire du programme débogué de manière à associer une sorte de « marque temporelle » à chaque objet alloué. Cette marque n'est pas une mesure de temps, elle a pour but de représenter une phase logique de l'exécution. Lorsque l'exécution est suspendue, l'utilisateur peut débuter une nouvelle phase à l'aide de la commande suivante :

```
(bugloo) (heap mark)
```

Ainsi, pour détecter les objets du tas qui ont été alloués par une partie du programme, il suffit de poser des points d'arrêt au début et à la fin de cette partie. Lorsque le premier point d'arrêt est atteint, l'utilisateur démarre une nouvelle phase a l'aide de la commande précédente puis reprendre l'exécution. Lorsque le deuxième point d'arrêt est atteint, il ne reste plus qu'à demander à l'inspecteur d'allocations d'afficher uniquement les objets présents dans le tas qui ont été créés à partir de la nouvelle phase temporelle.

5.2.4 Site d'allocation d'un objet

Lorsqu'un objet est alloué dans la machine virtuelle, le débogueur peut mémoriser le nom des fonctions qui sont présentes en sommet de pile. Plus tard, lorsque l'exécution sera suspendue, l'inspecteur d'allocation pourra retourner cette information pour tout objet présent dans le tas.

```
1  (module exemple1
2     (export (swapcons ::pair))
3     (main go))
4
5  (define (swapcons c)
6     (cons (cdr c) (car c)))
7
8  (define var '())
9  (define var2 (make-vector 11 0))
10
11 (define (go args)
12    (let* ((x (list 1 2 3))
13           (y (cdr x)))
14       (set! var y)
15       (vector-set! var2 1 y)
16       (print (swapcons (cons 1 2)))))
```

FIG. 5.2: Débogage mémoire sur un exemple

Le débogueur utilise les techniques de formatage générique (cf. chapitre 3) pour afficher correctement le sommet de pile mémorisé. Si l'utilisateur le souhaite, il peut aussi construire la vue virtuelle du sommet de pile (cf. chapitre 4). Cependant, cette construction ne s'opère pas au moment de la mémorisation du sommet de pile pour des raisons de performances. Ceci a deux conséquences :

- Les fonctions de haut niveau ayant une compilation complexe ne sont traitées qu'au moment de l'affichage. Si une fonction utilisateur est compilée en plusieurs fonctions JVM, le débogueur doit mémoriser toutes ces fonctions s'il veut les afficher correctement par la suite ;
- Lorsqu'un objet est alloué dans une bibliothèque, le sommet de pile contient le ou les fonctions de cette bibliothèque. Cela peut nuire à la pertinence de l'information mémorisée.

Il est nécessaire de mémoriser les `n` premières fonctions de la pile, mais il n'existe pas de valeur optimale pour `n`. Dans l'implantation, cette valeur peut être modifiée par l'utilisateur durant la session de débogage. Elle doit être un compromis entre précision des informations souhaitées et la pénalité mémoire qu'elle entraîne.

Le programme de la figure 5.2 présente un exemple d'information d'allocation. Supposons qu'un point d'arrêt ait suspendu l'exécution du programme dans la fonction `swapcons`, qui reçoit un doublet en paramètre et retourne un doublet dont les éléments sont inversés. L'utilisateur peut demander à l'inspecteur d'allocation d'afficher les fonctions responsables de l'allocation du doublet `c` en écrivant :

```
breakpoint 0 at (swapcons ::pair) in file alloc.scm:6
(bugloo) (info alloc c)
#0 pair MAKE_PAIR(Object, Object) in file foreign.java:2469
#1 (go ::pair) in file alloc.scm:11
```

On voit que l'objet a été alloué par une fonction de bibliothèque d'exécution nommée `MAKE_PAIR` écrite en Java. La fonction utilisateur responsable de la création de l'objet est la fonction `go` à la ligne 9. Cet exemple illustre bien le problème de pertinence de trace dû à la présence de fonction de bibliothèque. Un exemple complet d'inspection d'objet uniquement accessible depuis le tas sera présenté plus loin dans ce chapitre.

5.3 Inspecter les références entre les objets du tas

Cette section présente le deuxième outil de débogage mémoire développé dans BUGLOO. Le but de cet outil est de permettre d'inspecter le contenu des objets présents dans le tas, par exemple pour vérifier qu'un chaînage entre plusieurs objets est correct, ou bien pour vérifier

qu'aucune référence superflue n'existe dans le programme. L'outil est capable d'effectuer trois types d'opérations sur un objet donné du tas :

- il permet d'inspecter les références *sortantes* de l'objet, c'est-à-dire tous les objets que ce dernier référence à travers ses champs ;
- il permet de retrouver les références *entrantes* de l'objet, c'est-à-dire tous les objets du tas et les racines du GC qui le référencent ;
- il permet de déterminer quelles sont les racines du GC qui maintiennent l'objet vivant, en construisant des *chaînes de références entrantes* partant de l'objet et allant jusqu'aux racines.

Le débogueur fournit également un outil graphique permettant d'inspecter le tas « en deux dimensions » et de manière interactive. La suite de la section décrit les fonctionnalités d'inspection et leur intégration dans l'interface graphique.

5.3.1 L'inspecteur graphique de tas

L'inspecteur graphique est un outil écrit en Biglook [GS03b], une bibliothèque de programmation d'interfaces graphiques en Scheme. La figure 5.3 présente deux exemples d'inspection de références entre plusieurs objets du tas. La fenêtre graphique est décomposée en deux parties. Au sommet, une barre d'outil sert à régler l'affichage de l'inspecteur. On peut zoomer manuellement dans le graphe de références, ou bien utiliser un facteur de zoom automatique qui ajuste la boîte englobante du graphe inspecté à la taille de la fenêtre graphique. L'inspecteur de tas utilise la bibliothèque de production de graphe GraphViz [GKNV93]. Il est possible de choisir l'algorithme utilisé pour placer des objets dans le graphe :

- `dot` : cet algorithme aligne les objets du tas en strates horizontales. Chaque strate représente un niveau de référencement d'objet ; la figure 5.3(a) utilise cet algorithme ;
- `circo` : cet algorithme tire parti de la nature cyclique d'un graphe de références et positionne les objets du tas de manière à optimiser la présentation des cycles. La figure 5.3(b) utilise cet algorithme pour rendre le graphe présenté dans la figure 5.3(a).

Le reste de la fenêtre de l'inspecteur contient le rendu du graphe de références. Dans ce graphe orienté, les nœuds représentent les objets inspectés et les arêtes les références entre ces objets. La forme des nœuds dépend de leur nature :

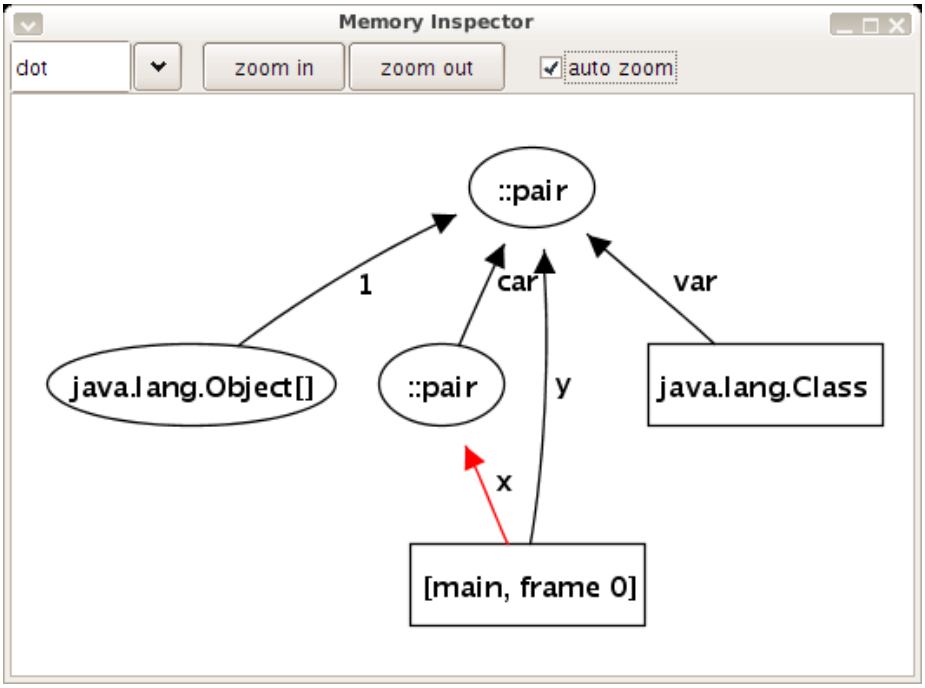
- un nœud ovale représente un objet ou un tableau présent dans le tas, le label associé au nœud indiquant le type cet objet ;
- Un nœud rectangulaire représente une racine du GC. Dans ce cas, le label indique la nature de la racine, comme par exemple une variable locale, ou un champ statique.

Dans la machine virtuelle Java, un objet ne peut pas « contenir » d'objet, il peut uniquement en référencer. Une arête partant d'un objet `a` et pointant un objet `b` indique que `a` possède une référence sur `b`. Chaque arête du graphe est associée à un label donnant des précisions sur la référence. Par exemple, lorsque `a` représente un objet, le label indiquera le nom du champs de `a` référençant `b`. En revanche, si `a` représente un tableau, le label indiquera l'indice du tableau qui référence `b`.

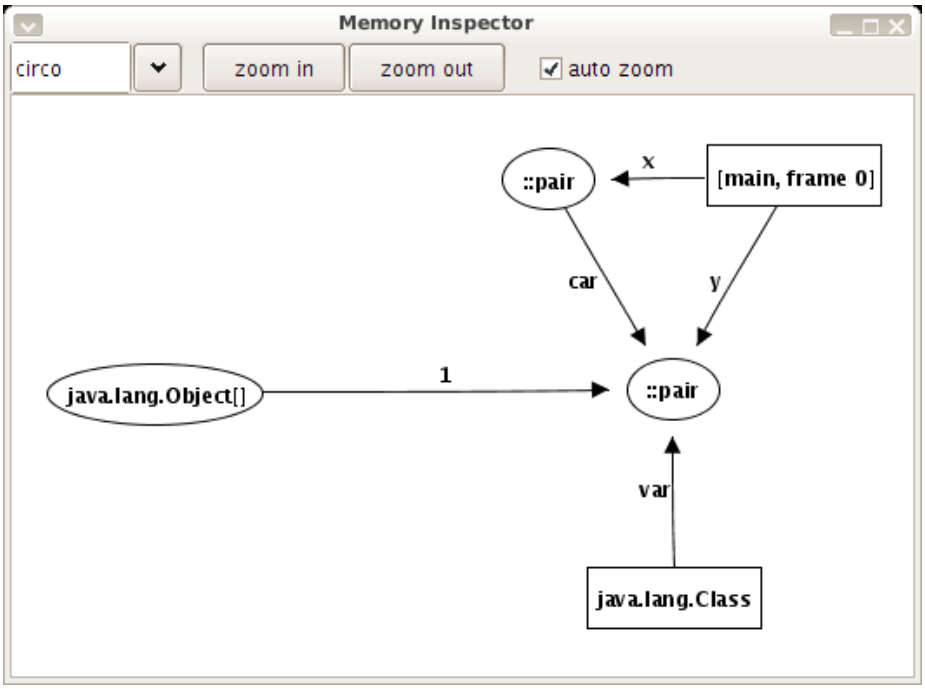
L'utilisateur peut ajouter des nœuds au graphe de références en se servant de la ligne de commande du débogueur :

```
(bugloo) (gui memory add name)
```

La commande précédente recherche l'objet référencé par une variable du programme appelée `name` et ajoute un nouveau nœud dans le graphe. La recherche de l'objet est effectuée à l'aide du mécanisme générique de recherche de variable (cf. chapitre 6). Cela permet de



(a) Chaînage rendu avec l'algorithme de disposition `dot`



(b) Le même chaînage rendu avec l'algorithme de disposition `circo`

FIG. 5.3: Exemple d'inspection graphique des objets du tas

rechercher les valeurs dans les différents niveaux lexicaux définis par les langages de haut niveau utilisés dans le programme débogué.

Lorsqu'un nouveau nœud ou une nouvelle arête est ajouté dans le graphe, ils apparaît en rouge. Dans la figure 5.3(a), on voit que la dernière opération sur le graphe a fait apparaître l'arête nommée `x`.

Chaque nœud du graphe dispose de son propre menu contextuel. Lorsqu'un nœud représente un objet présent dans le tas, une option du menu permet de visualiser l'objet dans l'inspecteur structurel (cf. chapitre 3). Les autres options du menu permettent d'afficher ou de masquer les références sortantes et entrantes du nœud, comme indiqué dans la section suivante.

5.3.2 Inspection des références des objets du tas

L'inspecteur de références permet d'analyser les liaisons qui existent entre différents objets du tas. Cet outil permet de visualiser les références que détient un objet sur d'autres objets du tas. Il permet aussi de retrouver tous les objets du tas détenant une référence vers un objet particulier. Les différentes opérations de l'inspecteur sont décrites ci-dessous.

Inspection des références sortantes

Les références sortantes d'un objet permettent de visualiser les objets du tas qui sont directement atteignable depuis cet objet. Dans la JVM, Il existe deux types de références directes : les références provenant des champs (statiques ou d'instance) d'un objet JVM, ou celles provenant d'un tableau d'objet JVM.

Les informations retournées par l'inspecteur de références dépendent de la nature de l'objet inspecté :

- si l'objet est un tableau d'objets JVM, l'inspecteur retourne tous les éléments non nuls du tableau ;
- si l'objet est une instance de classe JVM, l'inspecteur parcourt les champs d'instance contenant des objets et retourne toutes les valeurs non nulles ;
- si l'objet est une classe réifiée (c'est-à-dire une instance de la classe `java.lang.Class`), l'inspecteur parcourt les champs statiques de la classe réifiée contenant des objets et retourne toutes les valeurs non nulles.

Lorsque l'objet à inspecter est un objet virtuel, l'inspecteur renverra les objets pointés par ses champs virtuels. L'inspection masque donc automatiquement les détails de compilation d'une objet structuré *ad-hoc*.

Il est possible de masquer les références sortantes d'un objet lorsque cette information n'est plus désirée. Dans ce cas, tous les nœuds accessibles depuis cet objet sont automatiquement retirés du graphe de références.

Inspection des références entrantes

Les références entrantes d'un objet sont l'ensemble des pointeurs du tas dirigés vers cet objet. Cette information n'est pas directement conservée dans le tas de la JVM mais doit être calculée par le débogueur.

Connaître les références entrantes d'un objet peut se révéler très utile. Par exemple, certains algorithmes procèdent à des optimisations mémoire en partageant des objets dans le tas au lieu de les dupliquer. Ces optimisations entraînent souvent des bogues lorsque l'algorithme procède à des mutations sur les structures de données partagées. Ce genre de bogue est fréquent dans un langage de programmation comme Scheme, où l'on partage

souvent des doublets en mémoire. Connaître les références entrantes d'un objet permet de visualiser graphiquement le partage de structures de données et de s'assurer que les références entre les objets sont bien conformes aux attentes.

Pour obtenir la liste des références entrantes d'un objet `o`, le débogueur parcourt la liste des objets présents dans le tas en partant des racines du GC. À chaque fois qu'un objet ou une racine référence l'objet `o`, une nouvelle arête est insérée dans le graphe de références de l'inspecteur graphique. Une référence peut provenir :

- d'un champ d'instance d'un objet JVM. Dans ce cas, un nœud ovale représentant l'objet référant est inséré dans le graphe et une arête portant le nom du champ est créée ;
- d'un élément d'un tableau d'objet JVM. Dans ce cas, un nœud ovale représentant le tableau est inséré et une arête donnant l'index de la référence dans le tableau est créée ;
- d'un champ statique d'une classe JVM. Dans ce cas, un nœud rectangulaire représentant la classe référante est inséré et une arête portant le nom du champ statique est créée ;
- d'une variable locale dans la pile. Dans ce cas, un nœud rectangulaire indiquant le nom du *thread* (et le bloc d'activation) est inséré et une arête portant le nom de la variable est créée ;
- d'une référence native JNI. Dans ce cas, un nœud rectangulaire indiquant le type de référence est inséré, ainsi qu'une arête ne portant pas de nom ;
- d'un index dans le *Constant Pool* d'une classe JVM. Un *Constant Pool* est un descripteur privé provenant d'un fichier de classe JVM et contenant des constantes dont certaines (typiquement des chaînes de caractères) sont automatiquement transformées en objets par la JVM.

L'utilisateur peut masquer la liste des références entrantes d'un objet donné, ce qui a pour effet de supprimer du graphe tous les objets qui par récurrence référencent cet objet.

Contrairement aux références sortantes, il n'y pas de disposition particulière permettant de masquer la compilation des structures de données des langages de haut niveau : tous les objets retournés représentent des références physiquement présentes dans le tas.

Construction des chaînes de références inverses

Une chaîne de références inverses est une succession de références entrantes partant d'un objet du tas et terminant sur une racine du GC. Cette information permet de retrouver la ou les racines du GC qui maintiennent directement ou indirectement un objet en vie dans le tas. Par extension, cette information permet de comprendre la cause d'une fuite mémoire. Comme les références entrantes, les chaînes de références inverses sont des informations qui doivent être calculées par le débogueur.

Pour construire une chaîne de références inverses d'un objet `o`, le débogueur démarre la recherche en partant des racines du GC et suit tous les objets qu'il peut atteindre jusqu'à trouver l'objet cible `o`. L'algorithme utilise une simple recherche en profondeur, qui marque un objet comme étant vu (pour éviter de cycler durant la recherche) et qui cherche récursivement toutes ses références. Durant la recherche en profondeur, l'algorithme maintient une pile d'objet en cours d'exploration. Lorsque l'objet cible `o` est atteint, les objets présents dans cette pile forment la chaîne de références inverses.

Pour des raisons d'affichage, lorsque l'utilisateur se sert de la ligne de commande, le débogueur renvoie uniquement la première chaîne de références inverses qu'il a détectée.

En pratique, il est rare que plusieurs racines soient responsables d'une fuite mémoire. L'utilisateur conserve toutefois la possibilité d'afficher toutes les chaînes de références inverses en se servant de l'inspecteur graphique.

Comme pour les références entrantes, les chaînes de références inverses contiennent uniquement des pointeurs physiquement présents dans le tas. Il n'est pas possible de masquer la compilation de structures de données des langages de haut niveau.

5.4 Exemple de débogage mémoire

Cette section présente un exemple typique d'utilisation des outils de débogage mémoire précédemment décrit, afin de montrer comment l'utilisateur peut détecter les causes de bogues liés à des problèmes d'allocation.

Le programme présenté dans la figure 5.5 est une démonstration inclue dans la distribution de la bibliothèque graphique Biglook. Son but est d'illustrer les possibilités d'animation de la bibliothèque. Le programme définit un type d'objet graphique `bonhomme`, qui étend le type `canvas-image`. Il commence son exécution en chargeant six images (ligne 9) pour animer le bonhomme. Puis, la fonction `go` crée une fenêtre (ligne 15) composée d'un canevas graphique et d'un bouton. Lorsque l'utilisateur clique sur ce bouton, trente bonshommes sont disposés aléatoirement sur le sol et s'animent. Leur comportement est dicté par la fonction `new-runner` : un bonhomme se déplace plus ou moins vite vers la droite (lignes 32 à 34) et puis recommence quelques milli-secondes plus tard (ligne 35). Dès qu'il sort de la fenêtre, il est détruit (ligne 37) et un nouveau bonhomme est créé à gauche de la fenêtre (ligne 38).

Le programme affiche constamment trente bonshommes en mouvement. Toutefois, si le programme s'exécute pendant une longue période de temps, il se met à ralentir, puis finit par planter en indiquant que la mémoire est pleine. Cela indique clairement la présence d'une fuite mémoire.

Afin de détecter la cause du bogue, l'utilisateur doit inspecter le tas pour surveiller les allocations effectuées durant l'animation. Pour cela, il suffit de relancer le programme, d'attendre que la fenêtre graphique apparaisse et de suspendre l'exécution par un `CTRL+C`. À partir de ce moment, l'utilisateur pose une marque temporelle pour ne plus s'occuper des objets alloués avant l'animation, reprend l'exécution et clique sur le bouton de la fenêtre. Au bout d'un certain temps, il suspend l'exécution, force le GC à ramasser les objets morts et inspecte les objets encore vivants alloués depuis la marque temporelle.

La figure 5.6 présente les statistiques renvoyées par le débogueur. Plusieurs informations peuvent en être extraites. Tout d'abord, les six images sont stockées en mémoire par des objets du package `sun.java2d`. D'autre part, les trente bonshommes sont associés à trente

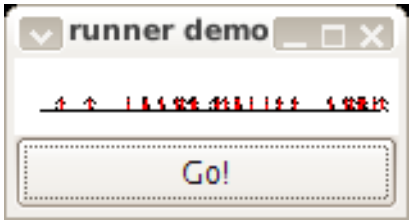


FIG. 5.4: Capture d'écran du programme débogué

```
1  (module anim2
2     (library biglook)
3     (static (class bonhomme::canvas-image
4                (images (default *bonhomme-image-list*))
5                (speed  (default (+ 10 (random 30))) read-only)))
6     (main go))
7
8  (define *bonhomme-image-list*
9     (let ((images (map (lambda (x) (file->image (format "runner s.xpm" x)))
10                       '(1 2 3 4 5 6))))
11       (set-cdr! (last-pair images) images)
12       images))
13
14 (define (go args)
15    (let* ((w (instantiate::window (title "runner demo")))
16           (c (instantiate::canvas (parent w) (height 30) (width  150))))
17       (instantiate::canvas-line (canvas c) (points '(10 20 140 20)))
18       (instantiate::button
19          (text "Go!") (parent w)
20          (command (lambda (_)
21                      (for-each (lambda (_)
22                                   (new-runner c (+ 10 (random 140))))
23                                (iota 30)))))
24       (widget-visible-set! w #t) ))
25
26 (define (new-runner c x0)
27    (let ((runner (instantiate::bonhomme (x x0) (y 15) (canvas c) )))
28       (define (run)
29          (with-access::bonhomme runner (image x images speed)
30             (if (< x 140)
31                 (begin
32                    (set! x (+fx 1 x))
33                    (set! image (car images))
34                    (set! images (cdr images))
35                    (after speed run))
36                 (begin
37                    (destroy runner)
38                    (new-runner c 10) ))))
39       (run)))
```

FIG. 5.5: Débogage d'un programme de démonstration de la bibliothèque Biglook

objets `javax.swing.Timer` pour gérer les animations. Parmi les 35 `pairs` allouées, on peut inclure les trente nécessaires au canevas pour stocker la liste de bonshommes affichés.

Dans la trace, une valeur ne semble pas conforme aux attentes : il y a 230 objets `bonhomme` (ainsi que leur implantation nommée `%canvas-image`) au lieu des 30 attendus. De manière aussi inattendue, on compte 230 objets `anim2`, qui représentent les fermetures créées lors de la création d'un `bonhomme` (ligne 28). Le grand nombre d'objets `java.lang.Object[]` inclut les 230 objets nécessaires à chacune de ces fermetures pour stocker les variables capturées `c` et `runner`[1].

L'information la plus étonnante est la présence de 25606 instances d'objets de classe `BJTimerAdapter` et `LinkedList$Entry`, qui à eux seuls occupent la moitié de la mémoire.

[1] L'implantation des fermetures Bigloo est décrit en détail dans le chapitre 6

```
(bugloo) (info heap mark)
java.util.LinkedList$Entry => 25607 instances (614568 bytes)
bigloo.biglook.peer.Jlib.BJTimerAdapter => 25606 instances (409696 bytes)
java.lang.Object[] => 256 instances (7416 bytes)
::bigloo.biglook.peer.Lwidget.%canvas-image => 230 instances (12880 bytes)
::bonhomme => 230 instances (7360 bytes)
anim2 => 230 instances (5520 bytes)
::pair => 35 instances (560 bytes)
javax.swing.Timer => 30 instances (1440 bytes)
javax.swing.Timer$DoPostEvent => 30 instances (480 bytes)
javax.swing.event.EventListenerList => 30 instances (480 bytes)
java.lang.Class => 17 instances (7952 bytes)
java.util.Hashtable$Entry => 6 instances (144 bytes)
sun.java2d.x11.X11CachingSurfaceManager => 6 instances (288 bytes)
sun.java2d.DefaultDisposerRecord => 6 instances (144 bytes)
::(array-of-char) => 3 instances (240 bytes)
::string => 2 instances (1424 bytes)
java.lang.InterruptedException => 1 instances (24 bytes)
java.util.LinkedList => 1 instances (24 bytes)
javax.swing.TimerQueue => 1 instances (16 bytes)
52401 instances in 44 classes (total 1073744 bytes)
2307936 bytes used in heap (total 5177344 bytes)
stats took 0.138s
```

FIG. 5.6: Exploration du tas du programme Biglook

Le programme n'a pas directement alloué ces objets. Il faut donc demander au débogueur de retrouver le responsable de ces allocations :

```
(bugloo) (info alloc ,(heap get "biglook.peer.Jlib.BJTimerAdapter" 0))
#0 (%after ::int ::proc) in class biglook.peer.Llib._after
#1 (after ::obj) in class biglook.Llib.after
```

Le débogueur nous indique que la fonction Biglook `after` est responsable de l'allocation des objets `BJTimerAdapter`. Cette fonction est appelée à peu près tous les dixièmes de seconde, ce qui explique le grand nombre d'objet dans le tas, mais n'explique toujours pas la fuite mémoire (on s'attendait à avoir 30 objets `BJTimerAdapter`). La fonction `after` doit aussi insérer ces objets dans une liste chaînée, étant donné le nombre quasi-identique d'objets `LinkedList$Entry` dans le tas. La liste est sûrement référencée par une racine du GC. Pour le vérifier, il faut demander au débogueur de calculer une chaîne de référence inverse :

```
(bugloo) (backref ,(heap get "::runner" 0))
#0 biglook.peer.Jlib.BJTimerAdapter
      | field data
#1 java.util.HashMap$Entry
      | field next
#2 java.util.HashMap$Entry
      | field next
#3 java.util.HashMap$Entry
      | field next
#4 java.util.LinkedList ====> class BJTimerAdapter : timer_list
```

La chaîne indique que la liste est stockée dans la variable statique `timer_list` de la classe `BJTimerAdapter` et que les instances de cette classe contiennent un pointeur vers une fermeture Scheme à exécuter. Cela explique pourquoi les fermetures `anim2` sont encore vivantes et pourquoi les objets `bonhomme` qu'elles ont capturés le sont aussi.

Pour supprimer la fuite mémoire, il faut maintenant chercher la fonction qui est censée retirer les objets de la liste chaînée `timer_list`. En examinant le code source, on trouve la fonction suivante :

```
public void actionPerformed( ActionEvent e ) {
    Object timer = e.getSource();
    if( thunk.funcall0() == foreign.BFALSE ) {
        ((javax.swing.Timer)timer).stop();
        timer_list.remove( this );
    }
}
```

Le déclencheur d'action asynchrone n'est retiré de la liste que si la fonction utilisateur renvoie la valeur Scheme `#f`. Ce comportement s'explique par la définition de `after` : lorsqu'on utilise cette fonction en remplaçant le temps en milli-secondes par le symbole `'idle`, la fermeture utilisateur est appelée à chaque fois que l'interface graphique est au repos. Les appels cessent lorsque la fermeture renvoie `#f`.

En conclusion, le débogueur nous a permis de localiser la cause de la fuite mémoire d'un programme d'exemple de la bibliothèque graphique Biglook. Pour la faire disparaître, il suffit de modifier la fonction `new-runner` pour qu'elle renvoie la valeur `#f` après la ligne 38. Sans un outil de débogage mémoire adapté, la localisation et la correction de ce bogue aurait sans doute été beaucoup plus compliquée, voir impossible.

5.5 Le profileur mémoire

Cette section présente le dernier outil de débogage mémoire mis au point dans Bugloo : un profileur d'allocation mémoire permettant à l'utilisateur d'étudier l'activité du GC durant l'exécution de ses programmes et de détecter des allocations qui ne seraient pas conformes à ses attentes. Cet outil est une adaptation de BMem, un profileur d'allocation dédié au langage Scheme. L'outil BMem permet de tracer les allocations et les ramassages qui ont lieu durant l'exécution. Il peut produire des statistiques tirées des traces enregistrées et les présenter dans des pages HTML qui seront étudiées une fois l'exécution terminée.

Le profileur fournit par Bugloo utilise BMem pour construire des statistiques HTML, mais adapte les mécanismes de trace pour permettre de profiler n'importe quel langage de haut niveau. La suite de cette section présente les principes de fonctionnement du profileur BMem et les informations qu'il récolte. Puis, nous décrivons les adaptations qu'il est nécessaire d'apporter au profileur pour tenir compte de la compilation complexe des langages de haut niveau vers une plateforme d'exécution généraliste comme la JVM. Enfin, nous présentons un exemple concret d'utilisation du nouveau profileur pour débogueur des problèmes d'allocation d'une bibliothèque de décompression d'archives ZIP écrite en Scheme.

5.5.1 Principe de fonctionnement du profileur BMem

Le profileur BMem compte les allocations effectuées par chaque fonction utilisée dans le programme débogué. De plus, il surveille l'évolution du tas : après chaque ramassage

du GC, le profileur enregistre la taille du tas, son taux d'occupation et le nombre d'objet alloués depuis le dernier ramassage.

BMem associe une table d'allocation à chaque fonction du programme débogué. Cette table contient le nombre d'objets que la fonction a alloués entre chaque ramassage, ainsi que leur type et leur taille en octet. Le principe de BMem est de fournir des traces *exactes*. En effet, contrairement aux profileurs comme gprof [GKM82] qui construisent leur statistiques grâce à des échantillonnages successifs de la pile d'exécution, BMem détecte toutes les allocations mémoire et met correctement à jour la table d'allocation de chaque fonction présente dans la pile à ce moment-là :

- La fonction en sommet de pile devient responsable d'une nouvelle allocation *directe*. Ce genre d'allocation permet de comptabiliser le nombre d'objet alloués par une fonction ;
- Une allocation *indirecte* supplémentaire est comptabilisée pour chacune des fonctions en attente dans la pile. Ce genre d'allocation permet de mesurer les allocations « cumulées » et donne un ordre de classement entre les fonctions. Les fonctions les plus présentes en piles auront un nombre plus élevé d'allocations indirectes. Par exemple, le point d'entrée du programme a toujours le plus grand nombre d'allocations indirectes.

BMem ne conserve pas l'état exact de la pile pour chaque allocation du programme. Toutefois, les allocations indirectes représentent une bonne approximation de cette information car les allocations cumulées permettent toujours de visualiser des fonctions sensibles du point de vue de l'allocation.

La figure 5.7 est une copie d'écran d'une page HTML construite à partir d'une trace BMem. La page présente les informations recueillies dans la trace sous différents angles. De haut en bas, on distingue :

1. les ramassages du GC qui ont eu lieu au cours de l'exécution,
2. les fonctions du programme classées par quantité d'allocations,
3. des statistiques sur le types des objets alloués.

Les éléments des graphiques peuvent être des numéros de ramassage du GC, des noms de fonctions ou des noms de types. Chaque élément est identifié par un code couleur. La signification des couleurs est précisée dans la légende à gauche de la page. Lorsque la souris s'arrête sur une partie colorée, le nom associé à cette zone colorée s'affiche dans une infobulle.

La première partie de la trace présente les informations sur les ramassages qui se sont produits dans le programme, ainsi que la mémoire totale alloué en Ko. Pour chaque ramassage, une barre horizontale indique le pourcentage d'occupation du tas par rapport à sa taille finale. La barre est décomposée en segments colorés. Dans le graphique de gauche, ces segments représentent la part de chaque fonction dans les allocations des objets vivants dans le tas. Dans le graphique de droite, les segments représentent le type des objets présents dans le tas et leur proportion.

La seconde partie de la trace est un classement des fonctions par allocation. Les graphes situés à gauche représentent les allocations *directes* des fonctions. Pour chaque fonction, une barre horizontale indique le pourcentage total de mémoire allouée. Selon le graphique, la barre est découpée en segments indiquant dans quels GCs ont eu lieu les allocations, la quantité de mémoire allouée par type ou le nombre d'objets alloués par type. Ainsi, la figure 5.7 montre que la fonction `alloc-wall` a consommé 41% de la mémoire totale, qu'elle a allouée autant de `pair` que de `vector`, mais que les objets `pair` n'occupaient qu'un quart de la mémoire.

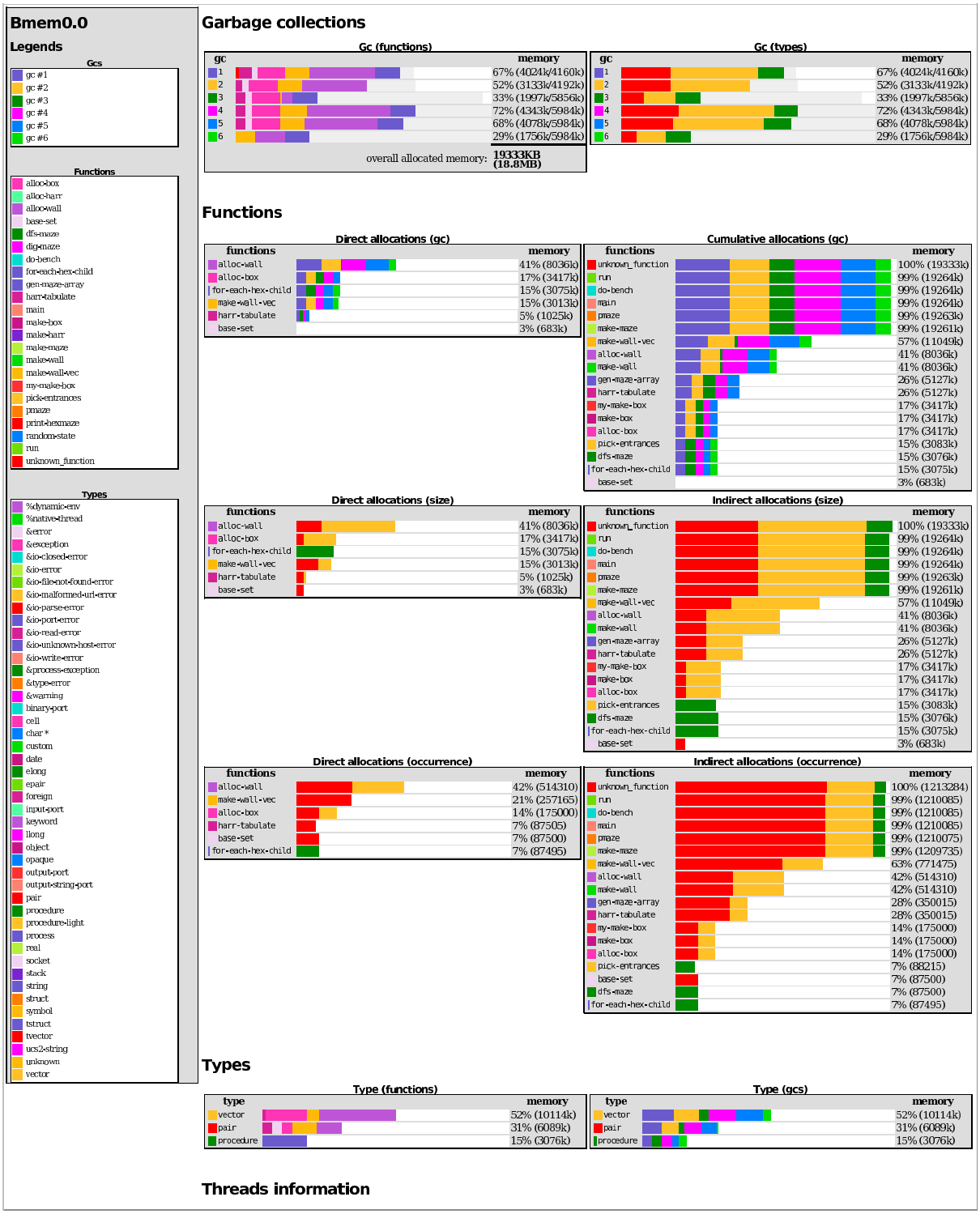


FIG. 5.7: Exemple de statistiques retournées par BMem

Les graphiques à droite de la trace représentent les allocations *indirectes*. Ils permettent de visualiser les fonctions qui ont été le plus souvent présentes dans la pile et dessinent en

générale une pyramide décroissante. Un grosse décroissance dans le graphique peut signifier qu'une fonction alloue plus que ce qui était attendu. Comme les graphiques précédents, les barres horizontales sont découpées en segments indiquant dans quels GCs ont eu lieu les allocations, la quantité de mémoire allouée par type ou le nombre d'objets alloués par type.

La dernière partie de la trace est un classement des objets les plus alloués dans le programme. Pour chaque type, une barre horizontale indique le pourcentage total d'occupation mémoire. Cette barre est découpée en segments indiquant la responsabilité de chaque fonction dans l'allocation du type (graphique de gauche), ou le pourcentage d'allocation du type pour chaque GC (graphique de droite).

5.5.2 Profilage des langages de haut niveau

BMem souffre de deux limitations majeures : il est conçu pour profiler uniquement des programmes Scheme compilés en code natif ; le programme cible doit être compilé en mode débogage (ainsi que les bibliothèques dont il dépend) pour être profilé correctement.

Le problème du profilage mémoire se complique lorsqu'il faut prendre en compte la compilation complexe des langages de haut niveau. En effet, les fonctions synthétiques qui implantent des fonctionnalités de haut niveau ne doivent pas apparaître dans la trace finale, sous peine de la noyer dans des détails de compilation nuisibles pour l'utilisateur. D'un autre côté, les allocations provenant de ces fonctions synthétiques doivent tout de même apparaître quelque part dans la trace si l'on veut présenter des informations non faussées à l'utilisateur.

Une manière de masquer les fonctions indésirables dans la trace finale est d'utiliser une adaptation de la méthode de construction de vue virtuelle de pile présentée dans le chapitre 4. Lorsqu'une allocation a lieu, une vue virtuelle de la pile est construite et les allocations directes et indirectes sont attribuées aux fonctions virtuelles. Dans cette approche, les allocations provenant de fonctions synthétiques doivent être attribuées aux fonctions utilisateur correspondantes, afin de réellement comptabiliser toutes les allocations qui se sont produites durant l'exécution.

5.5.3 Attribution des allocations des fonctions synthétiques

```
1  public class alloc extends function {
2
3      static function add_fun;
4
5      static Object add(Object x, Object y) {
6          Integer xi = (Integer)x;
7          Integer yi = (Integer)y;
8          return new Integer(xi+yi);
9      }
10
11     private int id;
12
13     Object funcall2(Object a1, Object a2) {
14         switch (id) {
15         case 0:
16             rt.trace=new trace("add",rt.trace);
17             Object ret=add(a1,a2);
18             rt.trace=rt.trace.next;
19             return ret;
20         }
21     }
22
23     public static void main (String args[]) {
24         add_fun = new alloc(0);
25         rt.print(add_fun.funcall2(1,1));
26     }
27 }
```

FIG. 5.8: Exemple de programme de haut niveau à profiler

Comme l'explique le chapitre 4, chaque fonction synthétique produite par le compilateur fait partie d'un motif de code servant à implanter une fonctionnalité du langage de haut

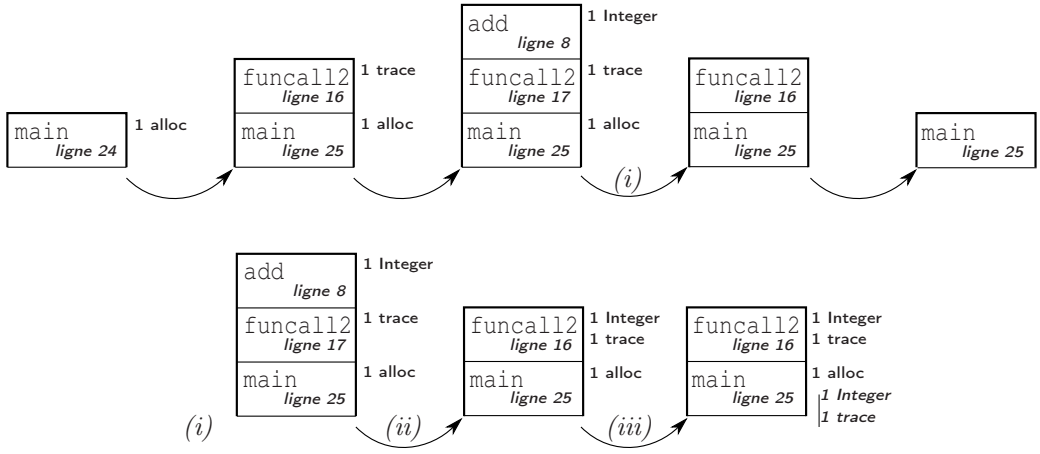


FIG. 5.9: Retarder le filtrage avant l'attribution des allocations

niveau. Les allocations effectuées par les fonctions synthétiques doivent donc être attribuées au fonctions logiques correspondantes dans le programme.

Lorsqu'une fonction synthétique alloue, il se peut que le motif auquel elle appartient ne soit pas encore *intégralement* présent dans la pile. Dans ce cas, l'attribution de l'allocation à la fonction logique correspondante est impossible, car la règle de remplacement en charge du motif ne peut pas encore être appliquée. La figure 5.8 illustre ce phénomène. Elle présente le résultat de la compilation d'un programme écrit dans un langage factice avec fonctions d'ordre supérieur. Dans ce langage, un appel de fonction est compilé en un appel à une fonction synthétique `funcall`n, qui elle-même rappelle la fonction utilisateur adéquate. De plus, cette fonction synthétique pose une de marque de débogage (ligne 15) avant d'effectuer la dispersion. Dans l'exemple, au moment où `funcall2` alloue, la pile virtuelle ne peut pas encore être construite car la fonction utilisateur `add` n'est pas encore présente dans la pile.

Le profileur peut construire ses statistiques seulement lorsque la vue virtuelle peut être reconstruite. Pour s'en assurer, l'idée est d'attendre que la taille de la pile atteigne un « maximum local » pour construire la vue virtuelle puis y attribuer les allocations directes et indirectes. Un maximum local est atteint juste avant de dépiler une fonction. La figure 5.9 décrit de manière schématique la construction de la trace virtuelle du programme de la figure 5.8. À l'étape 1, une allocation a lieu dans la fonction `main`. À ce moment, la pile n'a pas atteint un maximum local, l'allocation est conservée en mémoire et l'exécution est relancée. À l'étape 2, la fonction synthétique alloue un objet `trace` puis l'exécution continue. À l'étape 3, la fonction utilisateur `add` alloue un `Integer`. Lorsque cette fonction termine, le profileur détecte que l'étape 3 est un maximum local : il peut construire la vue virtuelle (*i*) qui attribuera les allocations directes (*ii*), puis y reporter les allocations indirectes (*iii*). Lorsque l'exécution reprend, la fonction `funcall2` termine. Cela marque un maximum local. Cependant, aucune allocation n'est reportée dans la vue, les précédentes ayant déjà été traitées. L'algorithme précis est décrit dans la section suivante.

Dans l'implantation actuelle du profileur, la construction de la vue virtuelle de pile s'effectue une fois que le programme a terminé son exécution pour pouvoir détecter aisément les maximum locaux. L'enregistrement de la trace est donc adapté : lorsqu'un objet est alloué, les indentifiants des fonctions présentes sur la pile sont sauvés sur disque. Lorsque le programme est terminé, la trace permet de reconstituer l'état exact de la pile pour chaque

$$\mathsf{profile}_T : [\mathsf{Function}] \times [\mathsf{Alloc}] \times [\mathsf{Rule}] \times [\mathsf{Alloc}] \longrightarrow [\mathsf{Function}]$$
$$\mathsf{set\_alloc} : [\mathsf{Function}] \times [\mathsf{Function}] \times [\mathsf{Alloc}] \longrightarrow [\mathsf{Function}]$$
$$\mathsf{add\_alloc} : \mathsf{Function} \times [\mathsf{Alloc}] \longrightarrow \mathsf{Function}$$

$$\frac{}{\mathsf{profile}_T(\emptyset, \emptyset, R, A) = \emptyset}$$

$$\frac{\mathsf{match\_stack}(\langle f_1 \ldots f_n, \omega, \emptyset \rangle) = \langle \mathit{false}, F, F' \rangle}{\mathsf{profile}_T(f_1 \ldots f_n, a_1 \ldots a_n, \langle \pi, \omega, \lambda_\rho \rangle.R, A) = \mathsf{profile}_T(f_1 \ldots f_n, a_1 \ldots a_n, R, A)} \quad (5.1)$$

$$\frac{}{\mathsf{profile}_T(f_1 \ldots f_n, a_1 \ldots a_n, \emptyset, A) = \mathsf{add\_alloc}(A, f_1).\mathsf{profile}_T(f_2 \ldots f_n, a_2 \ldots a_n, T, \emptyset)} \quad (5.2)$$

$$\frac{\mathsf{match\_stack}(f_1 \ldots f_n, \omega, \emptyset) = \langle \mathit{true}, f_1 \ldots f_i, f_{i+1} \ldots f_n \rangle \quad \lambda_\rho(f_1 \ldots f_i) = \langle \mathit{false}, \emptyset \rangle}{\mathsf{profile}_T(f_1 \ldots f_n, \langle \pi, \omega, \lambda_\rho \rangle.R, A) = \mathsf{profile}_T(f_{i+1} \ldots f_n, a_{i+1} \ldots a_n, R, a_1 \ldots a_i.A)} \quad (5.3)$$

$$\frac{\begin{array}{c}\mathsf{match\_stack}(f_1 \ldots f_n, \omega, \emptyset) = \langle \mathit{true}, f_1 \ldots f_i, f_{i+1} \ldots f_n \rangle \quad \lambda_\rho(f_1 \ldots f_i) = \langle \mathit{true}, g_1 \ldots g_i \rangle \\ \mathsf{set\_alloc}(f_1 \ldots f_i, g_1 \ldots g_i, a_1 \ldots a_i) = h_1 \ldots h_j \quad \mathsf{add\_alloc}(h_1, A) = h'_1\end{array}}{\mathsf{profile}_T(f_1 \ldots f_n, a_1 \ldots a_n, \langle \pi, \omega, \lambda_\rho \rangle.R, A) = h'_1.h_2 \ldots h_j.\mathsf{profile}_T(f_{i+1} \ldots f_n, a_{i+1} \ldots a_n, R, \emptyset)} \quad (5.4)$$

FIG. 5.10: l'algorithme de construction de la trace virtuelle

allocation.

5.5.4 L'algorithme de construction de la trace virtuelle

L'algorithme de construction de la trace virtuelle est une adaptation directe de la construction de la vue virtuelle de pile (cf. 4.4.1, page 52). L'algorithme $\mathsf{profile}_T$ construit la vue virtuelle selon un ensemble de règles T, en partant du sommet de la pile vers les fonctions les plus anciennes. Dans cette nouvelle utilisation, les blocs d'activation sont remplacés par des fonctions :

$$f \in \mathsf{Function} = \langle \mathit{name}, \mathit{args}, \mathit{ret}, \mathit{def}, \mathit{file}, \mathit{dalloc}, \mathit{ialloc} \rangle$$

Les cinq premiers éléments d'une fonction sont les mêmes que les cinq premiers d'un bloc d'activation : un nom (*name*), une liste contenant le type des arguments (*args*), un type de retour (*ret*), le lieu de définition de la fonction (*def*) et le nom du fichier source dans lequel est définie la fonction (*file*). Les deux derniers éléments servent à contenir toutes les allocations directes effectuées par la fonction f au cours du programme, ainsi que toutes les allocations indirectes dont elle est responsable. Ces deux éléments sont des listes de triplets de la forme $\langle g, t, s \rangle$ représentant des allocations : g est un entier indiquant la période du GC au cours de laquelle l'allocation a eu lieu, t est une chaîne de caractère représentant le type d'objet alloué et s est un entier représentant la taille de cet objet.

Un motif de compilation peut être détecté dans une trace en réutilisant la notion de filtre de bloc d'activation. Aucune adaptation n'est nécessaire puisque les éléments d'un filtre permettent de discriminer aussi bien des blocs d'activation que des fonctions.

La figure 5.10 décrit l'algorithme de construction de la vue virtuelle de pile (cf. chapitre 4.4) adapté au filtrage des traces. Comme expliqué dans la section 5.5.3, lorsque l'al-

gorithme doit construire une vue virtuelle, la pile représente un maximum local et chacune des fonctions qu'elle contient peut avoir alloué un ou plusieurs objets.

Dans l'algorithme $\mathsf{profile}_T$, le premier argument représente la liste des fonctions en attente dans la pile, notées $f_1 \ldots f_n$. Le second argument est une liste qui contient les allocations associées à chaque fonction de la pile. Dans l'algorithme, ces allocations sont notées $a_1 \ldots a_n$. Chaque a_i est une liste de triplet $\langle g, t, s \rangle$ (les mêmes que ceux conservés dans les champs *dalloc* et *ialloc*) représentant les allocations directes effectuées par la fonction f_i de la pile originale.

Le troisième argument est l'ensemble des règles de remplacement en cours de test sur le sommet de pile parmi toutes les règles dans T. Une règle de remplacement de trace est un triplet de la forme :

$$r \in \mathsf{Rule} = \langle \pi, \omega, \lambda_\rho \rangle$$

Comme les règles de remplacements de pile, l'élément π représente la priorité de la règle r, l'élément ω est un motif de pile qui détermine la séquence de fonctions à remplacer dans la trace et l'élément λ_ρ est la fonction à appliquer à cette séquence pour obtenir les fonctions virtuelles. Il n'y a pas d'élément λ_σ servant de substitution retardée car, contrairement à la vue virtuelle de pile, le profileur ne prend pas en compte les informations de lignes ou les variables locales dans la vue.

Enfin, le quatrième argument de $\mathsf{profile}_T$ représente des allocations en attente. Initialement vide, cet argument contient les allocations associées à des fonctions qui n'apparaissent pas dans la vue virtuelle et qui doivent donc reportées dans la prochaine fonction qui sera insérée dans la vue virtuelle. Cet argument joue un rôle similaire aux substitutions retardées (cf. 4.4).

Les différentes itérations possibles de l'algorithme de la figure 5.10 sont adaptées de l'algorithme de construction de vue virtuelle décrit au chapitre 5. Leur signification est détaillée ci-dessous.

cas 5.1 : si le motif de pile de la règle de remplacement courante ne s'applique pas, le processus continue en essayant la règle suivante.

cas 5.2 : si aucune règle de remplacement n'a pu s'appliquer, la fonction en sommet de pile est reportée telle quelle dans la vue virtuelle. Si des allocations étaient en attente, elles sont ajoutées à la liste des allocations directes de la fonction en sommet de pile à l'aide de la fonction add_alloc.

cas 5.3 : si le motif de pile de la règle de remplacement courante identifie le sommet de pile, le remplacement λ_ρ doit être appliqué. Si ce dernier renvoie un doublet dont le premier élément est la valeur *faux*, les fonctions détectées par le motif ω ne seront pas reportées dans la vue virtuelle. Toutefois, les allocations qu'elles auraient pu faire sont conservées dans la liste des allocations en attente. Cela permet globalement de prendre en compte toutes les allocations du programme.

cas 5.4 : si le motif de pile de la règle courante a identifié le sommet de pile et que le remplacement λ_ρ a retourné un doublet contenant la valeur logique *vrai*, les fonctions détectées doivent être remplacées dans la vue virtuelle. Pour cela, λ_ρ renvoie une liste $g_1 \ldots g_i$ contenant les fonctions virtuelles à insérer dans la vue.

Lorsqu'un motif de code est remplacé par une séquence de fonctions virtuelles, la fonction set_alloc se charge de reporter les allocations directes des fonctions synthétiques originales dans les fonctions virtuelles. La figure 5.11 détaille l'algorithme d'affectation des allocations directes. L'argument $\mathrm{L}_{\mathrm{orig}}$ est la liste des fonctions originales, l'argument $\mathrm{L}_{\mathrm{virt}}$ est la liste des fonctions retournées par le remplacement λ_ρ et l'argument $\mathrm{L}_{\mathrm{alloc}}$ est la liste

```
fonction set_alloc(L_orig, L_virt, L_alloc):
    L_orig ← f_1...f_n
    L_virt ← g_1...g_n
    L_alloc ← a_1...a_n
    result ← ∅
    L_tmp ← L_virt
    prev ← ⊥
    tantque prev = ⊥ faire
        prev ← tête(L_tmp)
        L_tmp ← reste(L_tmp)
    fintantque
tantque L_virt ≠ ∅ faire
    o ← tête(L_orig)
    v ← tête(L_virt)
    si v = ⊥ alors v ← prev
    add_alloc(v, tête(L_alloc))
    L_orig ← reste(L_orig)
    L_virt ← reste(L_virt)
    L_orig ← reste(L_orig)
    result ← result.v
    si v ≠ ⊥ alors prev ← v
fintantque
set_alloc ← result
```

FIG. 5.11: Attribution des allocations directes aux fonctions virtuelles

des allocations effectuées par les fonctions originales. La liste des g_i est de même longueur que celle des f_i et des a_i. Comme le montre l'algorithme de set_alloc en figure 5.11, la valeur d'un g_i détermine le remplacement effectué pour le f_i correspondant :

- si g_i est une fonction virtuelle, elle remplace f_i et prend les allocations directes a_i ;
- si g_i est la valeur $\perp$, f_i n'est pas reportée dans la vue virtuelle et les allocations directes sont attribuées à la fonction virtuelle la plus récente.

À titre d'exemple, considérons le remplacement suivant effectué par set_alloc :

$$\text{set\_alloc}(|f_1 \quad f_2 \quad f_3 \quad f_4|, |g_1 \quad \perp \quad g_2 \quad \perp|, |a_1 \quad a_2 \quad a_3 \quad a_4|)$$

Le résultat du remplacement renverra la séquence de fonctions virtuelles $|g_1 \quad g_2|$, sachant que les allocations directes a_1 et a_2 seront attribuées à g_1 et les allocations a_3 et a_4 le seront à g_2.

Dans l'algorithme profile_T, on peut voir dans le cas 5.4 que s'il y a des allocations en attente lorsque les fonctions virtuelles sont retournées, la première de ces fonctions est gratifiée de toutes les allocations en attente.

Enfin, si l'algorithme set_alloc permet de supprimer une fonction réelle dans la représentation virtuelle, il ne permet pas de la remplacer par plusieurs fonctions virtuelles. Il est donc à ce jour impossible d'employer cette technique de représentation virtuelle pour annuler l'effet de certaines optimisations du compilateur comme l'*inlining*.

5.5.5 Exemples de filtrages de pile

Cette section présente la syntaxe concrète des règles de remplacement de trace. Différents exemples sont présentés afin d'illustrer les principales utilisations de ces règles : regrouper des fonctions non désirables dans la trace, créer des fonctions virtuelles afin de masquer la compilation des langages de haut niveau et enfin masquer des fonctions jugées non utiles et différer leurs allocations.

Regrouper des fonctions inutiles

Lorsque l'utilisateur veut profiler une partie de son code, il faut qu'il puisse l'isoler dans la trace. En particulier, il ne souhaite pas forcément connaître le détail des allocations

effectuées dans des bibliothèques dont son programme dépend. À titre d'exemple, la règle de remplacement suivante permet de remplacer dans la trace toutes les fonctions provenant de la bibliothèque standard Java par une seule fonction virtuelle centralisant toutes leurs allocations :

```
(100
 [(+ (any any any ["^java\\..*$"]))]
 rep-java-runtime)
```

La règle précédente a une priorité arbitraire de 100, et détecte dans la pile une succession de fonctions dont la classe provient d'un package commençant par `java`. La fonction `rep-java-runtime` remplace les fonctions détectées par une seule fonction virtuelle qui contiendra toutes les allocations de ces fonctions :

```
(define unwanted-java #unspecified)

(define (filter-java-runtime methods)
   (unless unwanted-java
      (set! unwanted-java (create-unwanted-method "java" "RUNTIME")))
   (let ((res (make-list (length methods) #f)))
      (cons #t (cons unwanted-java (cdr res)))))
```

Dans la syntaxe concrète, le symbole $\perp$ est remplacé par la valeur Scheme `#f` (*faux*). La fonction `create-unwanted-method` renvoie une fonction virtuelle nommée `RUNTIME` provenant d'une classe fictive `java`. À chaque appel de la fonction de remplacement, un objet unique `unwanted-java` est retournée. Cet objet contient toutes les allocations directes et indirectes comptabilisées pour cette fonction depuis le début de la construction de la trace. Ces allocations sont automatiquement mises à jour par effet de bord par les fonctions set_alloc et add_alloc.

Masquer des motifs de code du compilateur

Les règles de remplacement conçues pour masquer des motifs de code synthétiques présents dans la pile peuvent être adaptées pour être utilisées par le profileur au moment de la construction de la trace virtuelle. Pour cela, il suffit en général de modifier légèrement la fonction de transformation. À titre d'exemple, le chapitre 6 décrit en détail le moyen de masquer l'implantation des appels d'ordre supérieur du langage Bigloo. La règle de remplacement suivante est une adaptation servant à la construction de la trace d'allocation :

```
(30
 [(["((?:BgL)*)(.*)"] (? any) any (? any))
  (["\\1_\\2"] ("bigloo.proc" . re-bigloo-args) "java.lang.Object" (\ 4))
  (["\\5"] re-bigloo-args "java.lang.Object" (\ 4))]
 rep-bigloo-high-order)
```

La fonction de transformation `rep-bigloo-high-order` est modifiée :

```
(lambda (methods)
   (let ((res (make-list (length methods) '())))
      (let* ((m (car methods))
             (c (dbg-method-class m))
             (n (dbg-method-name m))
             (key (string-append c n))
             (val (hashtable-get bigloo-funs key)))
         (unless val
            (hashtable-put!
             bigloo-funs key (create-closure-method m))
            (set! val (hashtable-get bigloo-funs key)))
         (cons #t (cons val (cdr res))))))
```

La fonction virtuelle renvoyée par la transformation est conservée dans une table de hachage contenant toutes les fermetures Bigloo déjà reconstruites dans la trace virtuelle. La clef de hachage est obtenue en concaténant le nom de la classe et le nom de la fonction originale. En utilisant cette règle, sur la pile ci-dessous, on obtient une représentation virtuelle sans fonctions synthétiques et qui préserve les allocations effectuées dans les fonctions utilisateur :

```
foo (2 pairs)
_foo                ⇝   foo (2 pairs)
funcall1                go (5 pairs)
go (5 pairs)
```

5.5.6 Retarder l'attribution des allocations

On emploie le système d'allocations retardées lorsqu'il faut masquer une fonction dans la pile afin de retrouver la véritable source d'une allocation. Par exemple, dans les programmes Bigloo, l'allocation d'un objet Scheme est traduite par le compilateur en un appel à une fonction de bibliothèque chargée de l'allocation. Par conséquent, la trace n'attribue pas les allocations directes aux fonctions utilisateurs, ce qui réduit grandement sa pertinence. La règle suivante permet de masquer l'implantation de l'allocation d'objets dans le compilateur Bigloo :

```
(100
 [(any any any "bigloo.foreign")]
 pending)
```

La fonction de transformation `pending` n'ajoute aucune fonction dans la pile virtuelle. Les allocations de la fonction de bibliothèque sont placées en attente et seront reportées dans la prochaine fonction qui sera insérée dans la pile virtuelle, c'est-à-dire une fonction utilisateur. Par exemple, lorsqu'on applique cette règles en plus de la précédente, la pile ci-dessous est correctement filtrée et l'allocation qui a lieu dans la fonction de bibliothèque est reportée dans la fonction utilisateur :

```
make_bint (1 bint)
foo (2 pairs)
_foo                ⇝   foo (2 pairs, 1 bint)
funcall1                go (5 pairs)
go (5 pairs)
```

5.6 Exemple d'utilisation : profilage du GZip Bigloo

Cette section décrit une utilisation concrète du profileur Bugloo. La bibliothèque d'exécution du compilateur Bigloo v2.8 intègre des fonctions de décompression des archives GZip [ZL77]. La figure 5.12 présente une utilisation de cette fonctionnalité. Le programme d'exemple ouvre un fichier passé en ligne de commande (ligne 5). Le port de lecture obtenu est passé à la fonction `open-input-gzip-port` qui renvoie un nouveau port de lecture (ligne 6) opérant comme un tube qui décompresse automatiquement les données avant de les renvoyer à l'utilisateur. Le programme lit sur ce nouveau port pour afficher le contenu du fichier ligne par ligne.

L'algorithme de décompression implanté dans la bibliothèque Bigloo est une réécriture en Scheme de l'outil de décompression `gzip`, écrit en C. Le compilateur Bigloo peut produire des exécutables natifs, JVM ou .NET, il réutilise donc le même code Scheme pour les trois générateurs de code. En testant les premières versions de la bibliothèque de décompression, il est apparu que les performances sur la plateforme JVM étaient inférieures à celles de la plateforme native, de manière plus importante que prévu.

Il a fallut profiler le programme d'exemple de la figure 5.12 pour voir si le problème venait des allocations. Une partie de la trace obtenue est affichée dans la figure 5.13. La trace montre que la majeure partie de la mémoire allouée est utilisée par des tableaux d'octet, ils représentent les chaînes de caractères renvoyées après une lecture dans le port Scheme. De manière inattendue, 76% des allocations — représentant un tiers de l'allocation globale — sont des objets `bint` servant à enrober des entiers pour pouvoir les conserver dans des champs de classes JVM devant contenir des objets. Cet enrobage n'est pas nécessaire en mode natif, car dans ce mode un champs Bigloo peut contenir des entiers ou des pointeurs. Les allocations supplémentaires observées dans la version JVM expliquent les baisses de performances inattendues.

La trace indique que les allocations de `bint` incombent à la fonction `liip` provenant de la bibliothèque de décompression :

```
(let liip ()
   (string-set! slide wp (string-ref slide d))
   (set! wp (+fx wp 1))
   (set! d (+fx d 1))
   (set! e (-fx e 1))
   (unless (=fx e 0) (liip)))
```

```
1  (module test-gzip
2     (main go))
3
4  (define (go args)
5     (let* ((p1 (open-input-file (cadr args)))
6            (p2 (open-input-gzip-port p1)))
7        (let loop ((x (read-line p2)))
8           (unless (eof-object? x)
9              (print x)
10             (loop (read-line p2)))))
11    #t)
```

Fig. 5.12: Utilisation de la bibliothèque de décompression de Bigloo

Cette fonction est une boucle affectant des variables capturées `wp`, `d` et `e`. Dans la JVM, ces variables doivent être enrobées dans des `bint` pour être stockées dans l'environnement de la fermeture `liip`. Chaque addition crée donc un nouvel objet `bint`. Pour éviter cela, il suffit d'effectuer la boucle sur des copies locales de ces variables capturées :

```
(let liip ((iwp::int wp) (id::int d) (ie::int e))
   (string-set! slide iwp (string-ref slide id))
   (if (=fx ie 0)
       (begin
          (set! wp iwp)
          (set! d id)
          (set! e ie))
       (liip (+fx iwp 1) (+fx id 1) (-fx ie 1)) ))
```

Les copies sont de véritables entiers, les additions effectuées durant la boucle n'allouent donc plus d'objet. Seule la sortie de fonction implique une création d'objet. Après un deuxième profilage, on constate que cette simple modification entraîne une réduction de plus de 95% les allocations de `bint` et de près d'un tiers les allocations globales du programme. De plus, on observe que le temps d'exécution du programme passe de 11.87s à 6.92s, soit un gain de vitesse de 42% !

En conclusion, le profileur s'est avéré être un outil de débogage mémoire efficace : il a permis de déterminer et de localiser la cause des ralentissements en très peu de temps ; de plus, en modifiant sept lignes dans une bibliothèque en contenant près de mille, il a permis des économies de mémoire et de temps substantielles.

Avant filtrage, la trace laissait apparaître les nombreuses fonctions de convertion de chaînes de caractères JVM (`String`) en chaînes Scheme (`byte[]`). De même, l'implantation des appels à la fermeture `liip` apparaissait dans les allocation indirectes (présence d'une

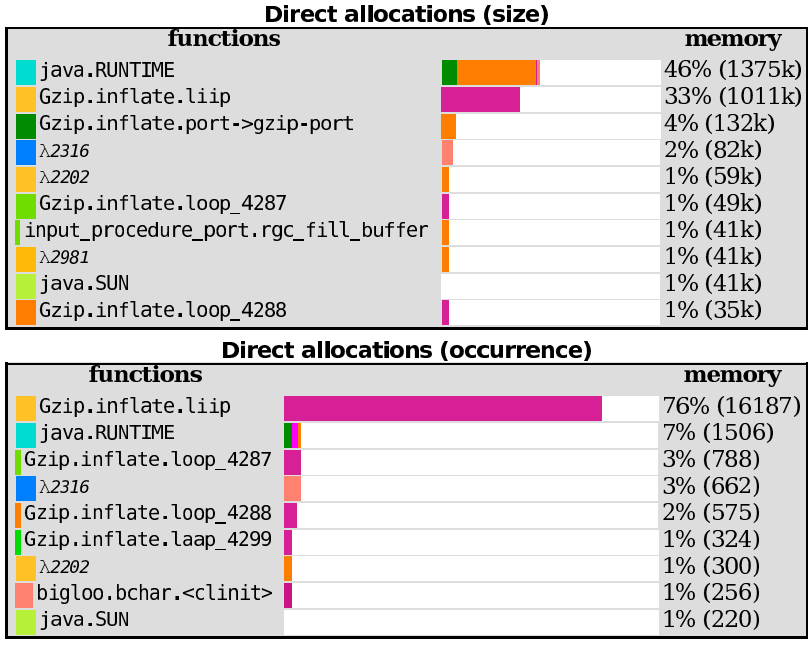


FIG. 5.13: Allocations directes durant la décompression

fonction `funcall`). Les techniques de filtrage de trace ont grandement amélioré la lisibilité globale des statistiques produites par le profileur.

5.7 Implantation des fonctionnalités de débogage mémoire

Dans cette section, nous décrivons l'implantation des outils de débogage mémoire, les principales difficultés de mise en œuvre et leurs limites actuelles.

5.7.1 Principes d'instrumentation

Pour implanter les fonctionnalités de débogage mémoire, le débogueur doit opérer deux types d'instrumentations dans la JVM de programme débogué : l'instrumentation de l'allocateur mémoire et l'instrumentation du code des classes JVM chargées dans le débogué.

Instrumenter l'allocateur mémoire

Lorsque la JVM du débogué est démarrée, elle charge en mémoire un module du débogueur écrit en code natif. Ce module utilise l'API JVMTI pour basculer l'allocateur de la JVM dans un mode spéciale « débogage ». Dans ce mode, lorsqu'un objet est alloué, un mot mémoire supplémentaire est automatiquement réservé dans le tas dans le but d'associer une « balise » à l'objet. Ce mot n'a pas de signification particulière pour la JVM. Il permet à l'agent de débogage de stocker l'adresse d'une structure qu'il a allouée pour contenir des informations de débogage mémoire associées à l'objet. Il est important de noter que l'accès à la balise d'un objet se fait en temps constant car l'objet et la balise sont consécutifs dans le tas. Ainsi, lorsque l'allocateur effectue un ramassage mémoire, la balise « bouge » avec l'objet.

Instrumentation des classes JVM

Lorsqu'une allocation a lieu, l'inspecteur d'allocation et le profileur mémoire doivent être avertis afin de traiter l'événement. La JVM ne fournit pas le moyen d'exécuter une fonction arbitraire lorsqu'une allocation a eu lieu. Pour y parvenir, le débogueur emploie une bibliothèque permettant d'instrumenter du code-octet à la volée : lorsqu'une classe est chargée dans la JVM, le code-octet de chacune de ses fonctions est modifiée de manière à ce que chaque allocation soit suivie d'un appel à l'agent de débogage. L'agent peut ainsi construire des informations de débogage et les associer à l'objet alloué.

Les appels et les retours de fonctions doivent tous être instrumentés pour que le profileur puisse maintenir une image des fonctions présentes en pile. Malheureusement, les fonctions de certaines classes primordiales de la JVM ne peuvent pas être instrumentées, comme par exemple les constructeurs de `java.lang.Class`. Cela force le débogueur à vérifier que son image de la pile est bien conforme à la réalité, ce qui occasionne un petit surcoût incompressible.

Lorsque le GC procède à un ramassage et qu'un objet mort est supprimé du tas, la JVM peut notifier l'agent, afin qu'il libère les informations de débogage associées à l'objet, comme par exemple les informations sur le site d'allocation.

5.7.2 Inspection du tas et site d'allocation

Les fonctionnalités de débogage mémoire peuvent être mise en œuvre efficacement grâce aux balises de l'API JVMTI, ces pointeurs supplémentaires alloués pour chaque objet lorsque l'allocateur mémoire de la JVM est en mode débogage. Par construction, JVMTI ne présente jamais l'adresse physique des objets JVM ou leur agencement mémoire. L'agent de débogage peut uniquement modifier la balise associée à un objet JVM. Par exemple, pour créer des statistiques sur l'occupation du tas, l'agent demande à JVMTI d'itérer sur tous les objets présents dans le tas. En retour, JVMTI passe à l'agent de débogage un pointeur sur la balise de chaque objet du tas. L'agent se sert de ce dernier pointeur pour mettre à jour les informations sur le nombre d'objets de cette classe et sur la place qu'ils occupent en mémoire. Lorsque la requête JVMTI est terminée, l'agent de débogage n'a plus la garantie que les balises resteront à leur place en mémoire du fait du GC générationel de la JVM.

Comme l'inspecteur d'allocation, l'inspecteur de références entrantes a besoin des balises sur objet. Il demande à JVMTI d'itérer sur toutes les références entre les objets présents dans le tas. À chaque itération, l'agent reçoit la balise de l'objet référant et celle de l'objet pointé. Lorsque la balise pointée représente l'objet cible, la balise référante est conservée. À la fin des itérations, l'agent demande à JVMTI de retrouver les objets associés à toutes les balises conservées et les renvoie au débogueur. Un processus similaire est mis en œuvre pour les chaînes de références entrantes : l'agent demande à JVMTI d'itérer sur toutes les références et maintient en parallèle une liste codant une chaîne de références. Dès que l'objet cible est détecté, la chaîne de références en cours est conservée.

Les résultats des inspections du tas sont renvoyés à travers une *socket* spéciale reliant le débogueur et le débogué. L'API de contrôle JDI et la couche de transport JDWP ne sont pas utilisées. Cela évite la création de tableaux d'objets pour retourner les résultats des inspections et permet de conserver de bonnes performances durant le débogage. En revanche, le résultat de l'inspection des références inverses transite par JDWP et alloue donc des objets intermédiaires. Leur quantité est négligeable et ne nuit pas aux performances du débogueur.

Les mécanismes d'inspection mémoire implantés sont peu invasifs, mais certains objets crées par l'instrumentation du débogueur sont visibles au moment de l'inspection mémoire. Par exemple, l'objet `java.lang.interruptedException` de la trace présenté en page 77 a été crée par le débogueur. En revanche, d'autres objets sont correctement masqués. Par exemple, lorsque l'agent recherche des références entrantes, il crée un tableau pour contenir le résultat de la recherche. Cet objet n'a pas de balise, cela permet donc de le différencier et de ne pas le prendre en compte lors de prochaines recherches de références entrantes.

5.7.3 Le profileur mémoire

L'implantation du profileur mémoire n'utilise pas les balises JVMTI (cf. 5.7.1). Le comportement de l'allocateur mémoire n'est donc pas modifié par rapport à une exécution normale. L'agent de débogage doit uniquement être notifié des allocations d'objets, des appels et des retours de fonctions.

Les informations sur l'état de la pile au moment d'une l'allocation sont directement sauvées sur disque. Chaque *thread* du programme peut avoir sa propre trace. À chaque fois que l'état de la pile est sauvé dans la trace, l'agent mémorise la taille courante de la pile. Lors de la prochaine allocation, seules les fonctions ayant changé dans la pile sont sauvées dans la trace. Initialement, la limite inférieure équivaut à la taille de la pile lors

de la dernière allocation. Cette limite est mise à jour automatiquement par l'agent lors des appels et des retours de fonctions.

L'algorithme de construction de trace virtuelle proposé ne peut pas masquer correctement certains motifs de code. Par exemple, il ne peut pas reconstruire de bloc d'activation pour une fonction ayant été *inlinée*. D'une manière générale, ce genre d'information nécessite de connaître l'information de ligne au moment de l'allocation. Il serait nécessaire de rajouter cette information dans les descripteurs d'allocations $\langle g, t, s \rangle$ (cf. 5.5.4, page 83). De même, lorsque l'algorithme est en présence de l'évaluateur du langage, il ne peut pas inférer la fonction interprétée car il n'a pas accès aux variables locales de l'évaluateur[2]. Un cinquième attribut sur utilisateur pourrait être sauvé dans le descripteur au moment de l'allocation. Cela pourrait servir à conserver une marque comme le nom de la fonction interprété en cours d'évaluation au moment de l'allocation.

Dans l'implantation actuelle, l'enregistrement de la trace sur disque est effectuée de manière naïve. En réalité, toutes les allocations dans le programme proviennent souvent d'un nombre de sites restreint dans le code source. Ainsi, dans la trace, si l'état de la pile à sauver a déjà été rencontré, il peut être remplacé par un identifiant unique. Cette optimisation réduirait considérablement la taille de la trace. De même, durant la construction de la trace, une même configuration de pile est transformée plusieurs fois. Il serait préférable de lire la trace entièrement afin de reconstruire un graphe d'appel complet, puis de construire la vue virtuelle de ce graphe. Cela éviterait les constructions redondantes de la vue virtuelle et apporterait un gain de temps très important.

Enfin, l'implantation actuelle à base d'enregistrement de trace sur disque a un défaut fondamental : elle produit des traces de taille proportionnelle au nombre d'allocations qui ont lieu dans le programme. Si cela n'est en général pas un problème pour surveiller la consommation mémoire d'un petit algorithme, cela devient impraticable pour des programmes traitant des très gros volumes de données et allouant beaucoup de mémoire.

5.8 Travaux reliés

JProfiler [Et04] est un profileur multi-fonctions fournissant des fonctionnalités de débogage mémoire. Il permet d'afficher en temps réel la consommation mémoire du tas d'un programme, de surveiller l'allocation mémoire entre deux points dans le temps, d'afficher les références sortantes, entrantes ou des chaînes de références inverses. Les outils développés dans Bugloo sont grandement inspirés du fonctionnement de cet outil. Toutefois, JProfiler ne permet pas de retrouver le site d'allocation des objets, ni d'enregistrer des traces d'allocation mémoire. De plus, Bugloo est un débogueur, ce qui accroît la souplesse d'inspection du programme débogué.

KBDB [SB00] est un outil capable d'exhiber les causes de fuites mémoires dans les programmes Bigloo compilés en code natif. KBDB utilise une version spéciale de l'allocateur mémoire Bigloo afin de conserver le site d'allocation des objets créés. Il permet d'afficher une chaîne de référence inverse pour un objet donné. Cette information n'est pas calculée à la demande comme dans Bugloo. Elle est conservée dans un champs supplémentaire de l'objet et mise à jour automatiquement lors d'un ramassage. La fonctionnalité est donc plus coûteuse que dans Bugloo. KBDB souffre d'une contrainte forte : pour qu'un programme puisse être débogué, il doit être compilé dans un mode spécial « débogage » incompatible avec le code compilé classique. Cela le rend peu pratique à utiliser.

[2] car l'algorithme manipule des traces et non des blocs d'activation encore présent sur la pile.

Jinsight [PS99, SDPK01] est un outil puissant de visualisation des références liant les objets du tas, pouvant être utilisé pour détecter des fuites mémoires. Il regroupe des séquences d'objets qui se référencent en se basant sur leur type. L'utilisateur peut marquer les objets du tas par âge. Lorsqu'un objet semble être retenu par le GC, il est possible de dérouler le graphe de références inverse pour retrouver la racine incriminée. Jinsight instrumente l'exécution du programme avec une API non standard. De plus, la quantité de donnée sauvegardée pour représenter le tas profilé est très importante et ne semble pas permettre le débogage de très gros programmes.

Hprof [O'H04] est un outil de profilage mémoire donnant des statistiques sur la consommation mémoire d'un programme. Contrairement aux profileur Bugloo, hprof construit sa trace en mémoire. À la fin de l'exécution, il affiche les différentes configurations de pile ayant alloué de la mémoire et permet de les classer par quantité. Cette présentation permet de trouver très rapidement les fonctions allouant trop de mémoire. Toutefois, les informations ne peuvent pas être triées comme dans Jinsight et leur quantité peut entraîner des problèmes pour les visualiser efficacement.

BMem est le profileur mémoire de programmes Scheme compilés en code natif, sur lequel le profileur Bugloo a calqué son fonctionnement. BMem construit ses statistiques d'allocation en RAM et nécessite que le programme à profiler soit compilé dans un mode spécial *débogage*. Ce mode maintient une liste des fonctions Scheme présentes dans la pile. À chaque allocation, cette liste est parcourue et les allocations directes et indirectes sont mises à jour. Bugloo est plus lent car il est obligé de sauver sa trace sur disque. Toutefois, il offre plusieurs avantages importants. Premièrement, il offre un mécanisme générique permettant de profiler tout type de langage de haut niveau. Deuxièmement, le programme à profiler n'a pas besoin d'être compilé spécialement, ce qui est très important pour une utilisation pratique. Troisièmement, le fait que le programme puisse être compilé normalement permet de conserver des optimisations lors de la compilation. Au contraire, le mode débogage de BMem réduit grandement les performances d'un programme natif comparé au mode de compilation classique. Enfin, Bugloo doit enregistrer l'état complet de la pile pour chaque allocation tout au long du programme. Il dispose donc du graphe d'allocation complet du programme. Ce surplus d'information peut être utilisé pour présenter des statistiques plus fines que les allocations indirectes. Toutefois, l'affichage du graphe doit être amélioré, notamment lorsqu'on est en présence de fonctions récursives ou co-récursives.

5.9 Conclusion

Ce chapitre a présenté les outils développés dans Bugloo d'aides pour localiser les causes des bogues liés à une mauvaise gestion de l'allocateur mémoire. Dans les plates-formes à désallocation automatique, ce type de bogue peut se manifester par des plantages dûs à des fuites mémoires ou par des problèmes de performance.

Pour lutter contre les fuites mémoires, Bugloo fournit un inspecteur d'allocation capable de déterminer le site d'allocation des objets. De plus, il permet d'inspecter graphiquement les références liant les objets du tas et d'exhiber les racines du GC responsables de rétentions d'objets. Pour améliorer les performances des programmes, Bugloo fournit un profileur d'allocations qui ne nécessite pas de compilation particulière. De plus, ce profileur est versatile puisqu'il accepte un ensemble de règles de remplacement pour permettre de déboguer correctement n'importe quel langage de haut niveau dont la compilation vers la JVM est complexe.

Plusieurs améliorations sont possibles dans les outils présentés. Tout d'abord, il serait souhaitable de perfectionner l'inspecteur des références inverses pour qu'il puisse masquer les pointeurs sur les objets intermédiaires produit par la compilation. Il serait nécessaire de modifier le profileur mémoire pour éviter les calculs redondants présents dans l'implantation actuelle. Enfin, les travaux sur le profileur devraient être poursuivis afin de mettre au point un mécanisme additionnel permettant de construire des informations correctes en présence de fonctions interprétées non JVM.

Deuxième partie

Spécialiser le débogueur pour les langages de haut niveau

Chapitre 6

Filtrage de la pile pour le langage Bigloo

Ce chapitre décrit en détail l'extension Bugloo développée pour permettre le débogage du langage Bigloo, une extension du langage fonctionnel Scheme. À travers une série d'exemples concrets, le chapitre donne un aperçu du travail que doit réaliser un implanteur de langage pour concevoir un support de débogage que tienne compte des fonctionnalités spécifiques d'un langage et qui permette de masquer tous les détails de sa compilation vers la JVM.

Ce chapitre est découpé en trois parties. La première partie illustre l'utilisation de l'API de programmation de Bugloo pour fournir de nouvelles fonctionnalités de débogage spécifique au langage Bigloo. La seconde partie illustre la mise en pratique des mécanismes de représentation logique de pile et de sauts virtuels afin de masquer les détails de compilation du langage Bigloo qui apparaissent dans la pile. Enfin, la troisième partie développe spécifiquement le masquage des détails d'implantation de l'interprète Scheme embarqué de la bibliothèque d'exécution de Bigloo. Elle montre ainsi qu'il est possible de déboguer des programmes mêlant code JVM et code *ad-hoc* de manière complètement transparente.

6.1 Extensions des fonctionnalités du débogueur

Cette section décrit les aménagements et les extensions apportées à Bugloo pour prendre en charge certaines spécificités du langage Bigloo. Elle présente un nouveau type de point d'arrêt. Elle montre comment utiliser l'interprète Scheme embarqué de la bibliothèque Bigloo pour mettre en œuvre des points d'arrêt conditionnels. Elle décrit la manière de formater les différents types de données du langage Bigloo. Enfin, elle présente une extension des informations de débogage permettant d'exécuter le programme caractère par caractère.

6.1.1 Utilisation de l'interprète embarqué

Bigloo permet de compiler du code source Scheme en code-octet JVM. Sa bibliothèque d'exécution permet aussi de charger dynamiquement un code source Scheme et de l'évaluer à l'aide d'un interprète Scheme embarqué. Grâce au support de débogage pour Bigloo, l'utilisateur a accès à cet interprète depuis le débogueur pour évaluer des expressions Scheme arbitraires tout au long de l'exécution. Cela se fait au moyen de la commande `bgl-eval` :

```
(bugloo) (bgl-eval <s-exp>)
```

Cette commande effectue un appel dans la JVM du débogué pour demander à l'interprète d'évaluer l'expression passée en argument. L'évaluation s'effectue dans l'environnement lexical du programme au moment où l'exécution a été suspendue. L'évaluation de cette expression est donc équivalente à l'évaluation de l'expression suivante :

```
(let <environnement lexical du debogue> <s-exp>)
```

De la même manière, lorsque le débogué est suspendu, l'utilisateur a la possibilité de démarrer une boucle d'interprétation dans la JVM du débogué s'il souhaite interagir directement avec le programme. Cela revient à évaluer l'expression suivante dans le débogué :

```
(let <environnement lexical du debogue> (repl))
```

La fonction `repl` démarre la boucle de lecture-évaluation-affichage classique d'un environnement Scheme. L'utilisateur peut ainsi accéder l'environnement lexical du programme au moment de sa suspension. En revanche, il ne peut pas modifier les variables de types primitifs (entiers ou doubles) car le débogueur est obligé de passer des copies de ces valeurs au `repl`.

6.1.2 Points d'arrêt supplémentaire pour Scheme

Le support de débogage pour le langage Bigloo met à disposition différents types de points d'arrêt, afin de refléter les spécificités du langage comme les fonctions de première classe ou les fonctions interprétées. La spécialisation des points d'arrêts est décrite ci-dessous.

Points d'arrêts fonctionnels

En Scheme, les fonctions sont des objets de première classe pouvant être référencées par des variables. Comme cela n'est pas le cas dans la JVM, le compilateur Bigloo doit mettre en œuvre un type *ad-hoc* de pointeur sur fonction[1]. Grâce au support de débogage pour Bigloo, il est possible de poser des points d'arrêt sur des variables contenant des fonctions. Pour illustrer cette fonctionnalité, considérons le programme suivant :

```
1 (module exmod (main go))
2
3 (define fun print)
4
5 (define (go args)
6    (fun args)
7    (set! fun (lambda (x) (display "fun: " x)))
8    (fun args))
```

Dans ce programme `exmod` la variable `fun` référence successivement la fonction `print` puis une fonction anonyme. L'utilisateur a la possibilité de poser un point d'arrêt spécial sur cette variable `fun` afin de suspendre l'exécution lorsque la fonction qu'elle contient est exécutée :

```
(bugloo) (closure-bp add exmod fun)
```

[1] Les détails d'implantation de ce type sont exposés plus loin dans ce chapitre.

Le programme sera donc suspendu une première fois dans la fonction `print` et une seconde dans la fonction anonyme.

Pour implanter ce type de point d'arrêt spécifique aux langage fonctionnels, le support de débogage Bigloo utilise l'API de BUGLOO. Il crée un point d'arrêt mémoire (*watchpoint*) pour surveiller le contenue de la variable `fun`. À chaque fois que la variable est modifiée, le débogueur inspecte sa nouvelle valeur pour retrouver la fonction JVM pointée, puis place un point d'arrêt classique sur cette fonction.

Points d'arrêts interprétés

Les fonctions construites lors du chargement dynamique de code source ne sont pas compilées en code-octet JVM mais utilisent le code-octet de Bigloo. Or, JVMTI ne permet pas de poser des points d'arrêt dans ces fonctions interprétées *ad-hoc*. Pour palier à cette limitation, le support de débogage Bigloo met en œuvre ses propres points d'arrêt. Ces derniers s'utilisent de manière identique aux points d'arrêt JVMTI grâce à la commande `eval-bp` :

```
(bugloo) (eval-bp add "file.scm" fun)
```

L'interprète a été légèrement modifié pour pouvoir « marquer » un code-octet lorsque l'utilisateur veut poser un point d'arrêt. Lorsqu'un code-octet marqué est atteint, l'interprète appelle une fonction de notification dans laquelle a été placé un point d'arrêt classique au démarrage de la session de débogage. Cela a pour effet de suspendre l'exécution comme s'il s'agissait d'un point d'arrêt classique.

L'interprète maintient une liste des fonctions contenues dans chaque fichier chargé dynamiquement. Lorsque l'utilisateur pose un point d'arrêt, le débogueur effectue un appel dans la JVM du débogué pour demander à l'interprète Bigloo de poser une marque dans la fonction cible. Si la fonction n'est pas encore chargée, l'interprète conserve ce point d'arrêt en attente jusqu'à ce que la fonction soit chargée en mémoire.

Attribut de point d'arrêt conditionnel

Les débogueurs symboliques modernes fournissent des points d'arrêts conditionnels. Toutefois, le langage utilisé pour exprimer la condition est en général un sous ensemble — pas très bien défini — du langage utilisé dans le code source, comme c'est le cas dans GDB avec C. D'autres débogueurs sont encore plus restrictifs, à l'image de EDebug [LaL94] qui ne peut poser qu'un point d'arrêt conditionnel à la fois.

Le support de débogage pour Bigloo propose un nouvel attribut (cf. 3.3, page 29) nommé `:cond` afin de mettre en œuvre des points d'arrêts conditionnels pour Scheme. Avec cet attribut, une condition est une fonction Scheme, ce qui permet de concevoir des conditions très expressives. À titre d'exemple, considérons l'écouteur d'événements graphique suivant :

```
1 (define (click-handler e::int)
2    (cond
3       ((= e 1) (print "button 1 pressed"))
4       ((= e 2) (print "button 2 pressed"))
5       (else (print "never mind"))))
```

Supposons que l'utilisateur recherche un bogue se produisant uniquement lorsque le bouton 1 de la souris a été enfoncé juste après le bouton 2. Il peut utiliser la capture de variable pour construire une *mémo-condition* :

```
(bugloo) (bp add listener 2
             :cond (let ((but2 #f))
                      (lambda (env)
                         (cond
                            ((and (= (ref env 'e) 1) but2)
                             (set! but2 #f) #t)
                            ((= (ref env 'e) 2)
                             (set! but2 #t) #f)))))
```

La condition reçoit en paramètre l'environnement lexical du programme débogué à l'endroit où la suspension a lieu. Contrairement à l'interprète embarqué, l'environnement n'est pas accédé de manière transparente. Lors de la création de la condition, BUGLOO étend la fermeture en définissant une fonction local `ref` servant à accéder a l'environnement à travers la variable opaque `env`. Cette approche est contraignante pour l'utilisateur car il ne peut pas accéder à l'environnement directement. Cependant, elle permet d'utiliser les propriétés de mémorisation des fermetures.

6.1.3 Modules de décodage et d'affichage

Cette section décrit la manière dont le support de débogage pour Bigloo se sert des différents mécanismes d'affichage et de formatage de BUGLOO pour présenter correctement les valeurs des identificateurs et des objets des programmes Bigloo.

Décodage et formatage

Le langage rationnel décrivant les identificateurs Scheme est plus vaste que celui utilisé par la JVM. Ainsi, la compilation Scheme vers JVM a recours à un encodage pour pouvoir représenter les identificateurs Scheme. À titre d'exemple, considérons le programme suivant :

```
1 (module tst (main *the-main*))
2
3 (define (*the-main* args)
4    (let ((f *the-main*))
5       (printf "f:  a, args:  a" f args)))
```

Lors de l'inspection de la pile d'exécution, le bloc d'activation représentant la fonction `*the-main*` est par défaut formaté par BUGLOO de la manière suivante :

```
#0  BgL_za2thezd2mainza2zd2(bigloo.pair) in file tst.scm:5
```

Le support de débogage pour Bigloo fournit un module de décodage et de formatage des identificateurs. Ainsi, le bloc d'activation précédent sera correctement décodé et affiché de manière préfixe :

```
#0  (*the-main* ::pair) in file tst.scm:5
```

Affichage des objets Bigloo

Le langage Scheme définit plusieurs fonctions standard d'affichage des objets. Le support de débogage pour Bigloo offre donc différents modules d'affichage :

```
(bugloo) (info displayer bigloo)
closure, write-circle, write, display-circle, display, default
```

Le fonctionnement de certains modules, comme `display` ou `write`, consiste à appeler dans la JVM du débogué des fonctions de la bibliothèque d'exécution Bigloo en charge du formatage. Hélas, ces fonctions de bibliothèque ne donnent pas d'information pertinente pour les variables contenant des fonctions. Par exemple, voici comment le module `write` affiche la valeur de la variable `f` du programme précédent :

```
f (::proc) = #<procedure>
```

Le module `closure` se sert de l'ordre particulier dans lequel les fonctions sont produites dans les classes Bigloo pour retrouver le nom de la fonction associée à la variable `f` :

```
f (::proc) = (*the-main* ::obj) in file tst.scm:3
```

L'ordre suffit pour déterminer le nom de la fonction. Cette information peut donc s'obtenir sans avoir à compiler les programmes en mode débogage.

Inspection des structures ad-hoc Bigloo

Certaines informations utiles au débogage disparaissent après la compilation des programmes Bigloo. Par exemple, la plateforme d'exécution Bigloo gère ses propre types structurés. Une structure Bigloo `s` contenant deux champs `a` et `b` sera compilée en une classe JVM de type `bigloo.struct` contenant deux champs : le premier sert à conserver le nom de la structure utilisateur (c'est-à-dire `s`) ; le deuxième est un tableau à deux éléments pour conserver les valeurs des deux champs `a` et `b`. Dans ce schéma de compilation, les noms des champs de la structure sont perdus. Le support de débogage de Bigloo permet de charger des informations de débogage externes au programme pour que le débogueur puisse reconstruire la structure durant l'inspection :

```
(set! *bgl-struct-fields-alist*
      (cons '(s (a b)) *bgl-struct-fields-alist*))
```

Pour rejouer aisément des sessions de débogage, ces initialisations peuvent être ajoutées au fichier de configuration `.bugloo` chargé automatiquement au démarrage du débogueur.

6.1.4 Exécution pas-à-pas par caractère

Bugloo peut être utilisé conjointement avec le compilateur Bigloo afin de fournir une exécution pas-à-pas spéciale dont la granularité est le caractère. En effet, grâce au système d'information de débogage en strate (cf. 3.4.1, page 31), le compilateur peut produire plusieurs informations de lignes dans une classe JVM, chacune dépendant d'une autre. La JVM ne fait aucune interprétation sémantique sur la valeur des informations produites.

Le support de débogage pour Bigloo en profite : le compilateur Bigloo a été légèrement modifié pour produire une première strate de débogage contenant des informations de caractères permettant de localiser précisément chaque s-expression du programme. Puis, une seconde strate est produite pour accueillir les informations de lignes.

Dans les classes Bigloo, la strate par défaut est celle qui contient les informations de lignes. Cela permet aux débogueurs traditionnels d'utiliser des informations de lignes correctes sans être perturbés par les informations de caractères. Dans Bugloo, l'utilisateur peut choisir la strate à utiliser (cf. chapitre 3) afin d'obtenir une exécution pas-à-pas avec par ligne ou par s-expression. Dans ce dernier cas, une marque graphique est insérée dans l'éditeur pour matérialiser la position de l'exécution à l'échelle du caractère.

6.2 Vue virtuelle pour les programmes Bigloo

Cette partie du chapitre présente une utilisation concrète des techniques de représentations virtuelles de la pile. Elle décrit un ensemble de règles de remplacement conçus pour masquer les détails de compilation du langage Bigloo.

Bigloo est un dialecte de Scheme permettant la compilation séparée et dont les abstractions du langage sont assez proches de celles de la JVM. Par exemple, un module de code utilisateur est compilé en une classe JVM. Une fonction utilisateur est compilée en une fonction JVM statique. Malgré cette compilation quasiment « directe » vers le code-octet JVM, certaines fonctionnalités du langage doivent être implantées à l'aide de structures de données et d'appels intermédiaires pour s'exécuter correctement dans la JVM. Cela inclue par exemple les fonctions d'ordre supérieur ou l'interprète de code dynamique.

Dans cette section nous décrivons le filtrage des principales fonctionnalités de Bigloo qui doivent être émulées pour s'exécuter dans la JVM. Nous présentons tout d'abord quelques exemples triviaux, puis nous détaillons le masquage des appels d'ordre supérieur, le masquage des fonctions génériques et le masquage de la gestion des exceptions.

6.2.1 Exemple de filtre trivial : le démarrage du programme

En Bigloo, n'importe quelle fonction utilisateur peut être employée comme point d'entrée d'un programme. Le compilateur produit automatiquement dans la classe principale du programme une fonction JVM `main` classique, qui enrobe les arguments de la ligne de commande dans une liste Scheme puis qui appelle une seconde fonction nommée `bigloo_main`.

```
1 (module hello (main foo))
2
3 (define (foo args)
4    (print "hello world"))
```

⇝

```
1  public class hello extends proc {
2      {...}
3
4      public static Object foo() {
5          runtime.print("hello world");
6          return runtime.UNSPECIFIED;
7      }
8
9      public static void bigloo_main(pair argv) {
10         module_initialization();
11         foo(argv);
12     }
13
14     public static void main(String[] argv) {
15         bigloo_main(runtime.wrap_args(argv));
16     }
17 }
```

La fonction `bigloo_main` exécute les expressions situées dans l'environnement global du programme puis appelle la fonction utilisateur servant de point d'entrée. Une règle de remplacement (cf. 4.4.1, page 52) adéquat permet de masquer ces deux fonctions dans la pile :

```
(100
 [("bigloo_main" any any (? any))
  ("main" ("java.lang.String[]") "void" (\ 1))]
 mask)
```

La règle précédente indique que les fonctions `main` et `bigloo_main` doivent provenir de la même classe JVM, quel que soit son nom (`hello` dans l'exemple). La priorité 100 est arbitrairement choisie comme priorité par défaut.

6.2.2 Masquer une fonctionnalité du système : les appels d'ordre supérieur

Cette sous-section explique comment masquer l'implantation des appels d'ordre supérieur des fonctions Bigloo, notamment les structures de données synthétiques et les appels de fonctions intermédiaires produites par le compilateur. En Scheme, les fonctions sont des objets de première classe, pouvant être passés en paramètre, retournés comme valeurs et sauvés en mémoire. Comme cela n'est pas possible dans la JVM, le compilateur doit implanter son propre type abstrait « pointeur sur fonction » [SS02] : un programme Bigloo est composé de modules, dans lesquels on attribue aux fonctions (des fermetures) un numéro unique. Lors de l'exécution, les fonctions et les fermetures fraîchement allouées sont représentés par un objet JVM de type `proc` contenant le numéro unique de la fermeture, son arité et ses variables capturées. Les appels de fonctions d'ordre supérieur sont implantés par un appel à une fonction de dispersion définie dans l'objet `proc`, qui elle même appelle la bonne fonction utilisateur.

Appels de fonctions d'ordre supérieur

Pour montrer le problème de pollution qui apparaît dans la pile du fait de l'implantation particulière des appels d'ordre supérieur, examinons le programme suivant :

```
1 (module exemple1 (main go))
2
3 (define (plus2 x::int)
4    (+ x 2))
5
6 (define (go args)
7    (let ((f plus2))
8       (print (f 10))))
```

Ce code définit un programme `exemple1` qui commence dans la fonction `go`. Supposons que l'exécution de ce programme soit suspendue à la ligne 4. On peut demander à Bugloo les fonctions présentes dans la pile brute — c'est-à-dire avant filtrage — à ce moment précis en faisant :

```
(bugloo) (info stack)
#0  (plus2 ::int) in file exemple1.scm:4
#1  (_plus2 ::proc ::obj) in file exemple1.scm
#2  (funcall1 ::obj) in file exemple1.scm
#3  (go ::pair) in file exemple1.scm:8
#4  (bigloo_main ::obj) in file exemple1.scm
#5  (main ::(array of java.lang.String)) in file exemple1.scm
```

La variable `f` située à la ligne 7 est l'objet de type `proc` qui représente la fonction `plus2`. On voit que `go` appelle la fonction synthétique `funcall1` (une méthode JVM de l'objet `proc`) pour émuler un appel d'ordre supérieur d'arité 1. L'argument de `funcall1` est le nombre 10 dans l'exemple. Puis, `funcall1` renvoie l'appel vers la fermeture `_plus2`, qui est un adaptateur servant à s'assurer que l'argument utilisateur est bien un entier avant d'appeler la vrai fonction `plus2`. Le premier argument de `_plus2` contient la valeur des variables capturées. Cet argument n'est pas conservé lors de l'appel à `plus2` car cette fonction est globale et ne capture pas de variable. Dans cet exemple, les fonctions dans les

blocs 1 et 2 sont générées par le compilateur. La règle de remplacement suivante montre comment on peut se servir des références du langage *Omega* pour modéliser une sorte de détection conditionnelle :

```
(100
 ([("((?:BgL)?)(.*)"] (? any) any (? any))
  (["\\1_\\2"] ("bigloo.proc" . re-bigloo-args) "Object" (\ 4))
  (["\\5"] re-bigloo-args "Object" (\ 4))]
 sf-bigloo-high-order-procedure-call)
```

Le premier bloc d'activation dans le motif capture le nom de la fonction courante dans la pile, ainsi que le type de ses arguments et la classe dans laquelle elle est définie. Le nom de la fonction est décomposé en deux sous-sauvegardes : un préfixe pouvant être `BgL` (si le nom de la fonction est encodée) ou rien et une deuxième partie comprenant le reste du nom. Le filtre de bloc suivant dans le motif stipule que la fonction suivante dans la pile doit être définie dans la même classe que la première fonction (`(\ 4)` dans la règle) et que son nom doit être le même et préfixé d'un tiret supplémentaire. Le premier argument de cette fonction doit être de type `proc`. Les autres arguments sont testés au moyen de la fonction utilisateur `re-bigloo-args` : les adaptateurs produits par Bigloo reçoivent et renvoient toujours des objets de type `obj`, donc `re-bigloo-args` vérifie seulement que les n arguments restant soient du type `obj` et qu'il y ait autant d'argument que dans la fonction `plus2`. Avant de retourner, `re-bigloo-args` détermine dynamiquement le nom que doit avoir la prochaine fonction dans la pile et l'ajoute dans la mémoire (`\5` dans la règle) : ce doit être `funcall`n si $n < 4$, ou `apply` sinon[2]. Le même type de test sur les arguments est répété dans le troisième filtre de bloc de la règle.

La fonction `sf-bigloo-high-order-procedure-call` est définie par `(lambda (x) (list (car x)))` : elle reçoit la liste des trois blocs d'activation détectés et renvoit le premier dans la vue virtuelle. Lorsqu'on utilise cette règle en plus de celle décrite en section 6.2.1, on obtient la vue virtuelle suivante :

```
(bugloo) (info stack filter)
#0  (plus2 ::int) in file exemple1.scm:4
#1  (go ::pair) in file exemple1.scm:8
```

Appel d'ordre supérieur à des fermetures

Les appels d'ordre supérieur sont compilés différemment lorsque la fonction appelée n'est pas globale, c'est-à-dire lorsqu'il s'agit d'une fonction interne ou anonyme. En effet, une fonction globale est compilée en deux fonctions : un adaptateur qui vérifie le type de ses arguments et une autre fonction qui produit les vrais calculs. Quand la fonction appelée peut être trouvée statiquement, le test de typage est effectué à la compilation et l'appel à l'adaptateur est évité. Il n'en va pas de même pour les appels d'ordre supérieur : par exemple, dans un module, lorsqu'on compile un appel à une fermeture provenant d'un module différent, le compilateur n'est plus en mesure d'effectuer le test de typage. C'est pour cette raison que les fonctions anonymes sont compilées en une seule fonction JVM faisant à la fois le test de typage et les vrais calculs. Pour illustrer cette particularité, considérons le programme suivant :

[2] cela reflète une optimisation du compilateur Bigloo pour accélérer les appels d'ordre supérieurs de moins de quatre arguments.

```
1 (module exemple2 (main go))
2
3 (define (make-anon y)
4    (lambda (x)
5       (+ x y)))
6
7 (define (go args)
8    (let ((a (make-anon 10)))
9       (print (a 20))))
```

Le programme fait un appel à la fonction `make-anon` pour créer une nouvelle fermeture, puis fait un appel à cette dernière. Supposons que l'exécution soit suspendue à la ligne 5. La figure 6.1 présente l'état de la pile d'exécution et celui des variables locales au moment de la suspension :

```
(bugloo) (info stack)
#0  (<anonymous:1006:exemple2.scm:4> ::proc ::obj) in file exemple2.scm:5
#1  (funcall1 ::obj) in file exemple2.scm
#2  (go ::pair) in file exemple2.scm:9
#3  (bigloo_main ::obj) in file exemple2.scm
#4  (main ::(array of java.lang.String)) in file exemple2.scm

(bugloo) (info args)
<env:1:y> (::exemple2) = <anonymous:1006:exemple2.scm:4> in file exemple2
x (::bint) = 20
```

(a) vue virtuelle sans filtrage de la fermeture

```
(bugloo) (info stack filter)
#0  (<anonymous:1006:exemple2.scm:4> ::obj) in file exemple2.scm:5
#1  (go ::pair) in file exemple2.scm:9

(bugloo) (info args filter)
y (::bint) = 10
x (::bint) = 20
```

(b) vue virtuelle tenant compte de la fermeture

FIG. 6.1: Contenu de la pile lors d'un appels à une fermeture

Dans la trace de la figure 6.1(a) on voit que l'appel de fonction d'ordre supérieur n'utilise pas d'adaptateur. On remarque que la fonction anonyme contient un argument synthétique de type `proc` qui contient l'identifiant de la fermeture et son environnement. Lorsqu'on liste les variables locales JVM, on voit apparaître l'objet synthétique nommé `<env:1:y>`[3]. La variable capturée `y` n'est pas présente dans cette liste, mais sa valeur est conservée dans l'objet synthétique. La règle de remplacement suivante montre comment on peut utiliser le concept de bloc d'activation virtuel (présenté au chapitre 4.4.1, page 52) pour masquer ces détails de compilation :

[3]son type (`exemple2`) est une sous-classe de `proc`. Voir [SS02] pour les détails de la compilation des fonctions.

```
(100
 [(any ("bigloo.proc" . (? re-bigloo-args2)) "Object" (? any))
  (["\\2"] (\ 1) "Object" (\ 3))]
 sf-bigloo-high-order-closure-call)
```

Le motif de bloc est similaire à celui employé pour masquer les appels d'ordre supérieur aux fonctions globales. La fonction `re-bigloo-args2` capture les arguments de la première fonction, les compte et met à jour la mémoire pour fixer le nom de la prochaine fonction à détecter dans la pile. De plus, ces deux fonctions consécutives dans la pile doivent provenir de la même classe JVM. Le premier bloc d'activation ne peut pas être reporté « tel quel » dans la vue virtuelle. La fonction `sf-bigloo-high-order-closure-call` se charge de retourner un bloc d'activation virtuel qui soit un clone de ce bloc sans l'argument synthétique. La figure 6.1(b) présente l'état de la pile après application de la règle. On voit que le bloc virtuel 0 filtre la variable synthétique `<env:1:y>` et qu'elle fait apparaître `y` comme s'il s'agissait d'une variable locale classique.

6.2.3 Masquer une fonctionnalité de bibliothèque : les fonctions génériques

En Scheme, il est fréquent d'utiliser du code de bibliothèque pour implanter des fonctionnalités de niveau « langage ». Cela s'explique par l'aisance à manipuler du code comme des données grâce aux S-expressions et au système de macros. Par exemple, dans le cas de Bigloo, la bibliothèque standard fournit des fonctions polymorphes par sous-typage similaires aux fonctions génériques de CLOS [BDG+88]. Une fonction générique Bigloo est un conteneur de fonctions spécifiques appelées *méthodes*. Quand la fonction générique est appelée, elle choisie une méthode à exécuter en fonction du type du premier paramètre de l'appel. Cette fonctionnalité n'est pas codée « en dur » dans le compilateur, elle est implantée grâce à des macros. Malheureusement, la structure logique de fonction générique et de méthode est perdue après la phase de macro-expansion. Voici un exemple illustrant ce problème :

```
1  (module exemple3
2     (export
3      (class a
4         field::int)
5      (class b::a))
6     (main go))
7
8  (define-generic (fun obj::a i::int)
9     (+ (a-field obj) i))
10
11 (define-method (fun obj::b i::int)
12    (* (b-field obj) i))
13
14 (define (go args)
15    (print (fun (instantiate::b (field 10)) 20)))
```

Le programme précédent définit une classe `a` et une classe `b` qui hérite de `a`. Il définit ensuite une fonction générique `fun` à la ligne 8 qui prend en paramètre un arguent de type `a` et un de type `int`. À la ligne 11, la fonction générique est spécialisée pour les arguments

de type `b`. Par la suite, l'appel à `fun` exécutera la méthode définie à la ligne 11. Voici l'état de la pile brute lorsque l'exécution atteint la ligne 12 :

```
(bugloo) (info stack)
#0  (fun-b_3_1049 ::proc ::obj ::obj) in file exemple3.scm:12
#1  (funcall2 ::obj ::obj) in file exemple3.scm
#2  (fun ::a ::int) in file exemple3.scm:8
#3  (go ::pair) in file exemple3.scm:15
#4  (bigloo_main ::obj) in file exemple3.scm
#5  (main ::(array of java.lang.String)) in file exemple3.scm
```

Beaucoup de détails de compilation apparaissent dans la pile. Tout d'abord, nous voyons que l'appel de fonction générique est implanté par un appel de fonction à `fun` (bloc 2), qui elle-même fait un appel d'ordre supérieur à la bonne méthode (bloc 0). On voit ensuite que la fonction dans bloc 0 qui représente la méthode générique a un nom encodé qui contient le nom du type pour lequel a lieu la spécialisation. Enfin, on voit que cette fonction est traitée comme une fonction anonyme par le compilateur et qu'en tant que telle, elle a perdu l'information de typage pour ses arguments et son type de retour.

Les trois premiers blocs d'activation de la pile doivent être remplacés par un bloc d'activation représentant la méthode définie à la ligne 11. Ce travail est une exemple typique où une règle de remplacement doit réunir des informations disséminées à travers la pile pour pouvoir reconstruire une information logique correcte.

```
(90
   [(re-bigloo-gen-name ("bigloo.proc" . (? re-bigloo-args2))
                        "Object" (? any))
    (["\\3"] (\ 2) "Object" (\ 4))
    (["\\1"] any any (? any))
    (? (["\\3"] (\ 2) "Object" (\ 5)))]
   sf-bigloo-generic-call)
```

Dans le premier filtre de bloc, la fonction `re-bigloo-gen-name` décode le nom du premier bloc d'activation détecté : il utilise le nombre entre les deux tirets comme décalage pour extraire le vrai nom de la fonction générique (c'est-à-dire `fun`) et pour le sauver dans la mémoire de capture. Elle sauve aussi la partie représentant le nom du type servant à la surcharge (c'est-à-dire `b`). Le filtre de bloc suivant est similaire à ceux utilisés pour masquer les appels aux fonctions anonymes. Ensuite, le troisième filtre de bloc vérifie que le nom de la troisième fonction du groupe est égale au nom extrait de la fonction en sommet de pile. Le quatrième filtre est facultatif : au cas où le code ait effectué un appel d'ordre supérieur, il permet de s'assurer que l'intégralité du motif soit masqué.

La fonction `sf-bigloo-generic-call` utilise la valeur des différents attributs capturés pour reconstruire un bloc d'activation virtuel représentant la méthode « logique » présente dans le fichier source : le nom et la signature du bloc d'activation virtuel sont similaires à ceux du troisième bloc, sauf pour le premier argument qui est égale à la deuxième valeur mémorisée. De plus, l'information de ligne utilisée provient du premier bloc. La pile virtuelle finale est présentée ci-dessous :

```
(bugloo) (info stack filter)
#0  (fun ::b ::int) in file exemple3.scm:12
#1  (go ::pair) in file exemple3.scm:15
```

Cette règle de remplacement soulève le problème de la priorité des règles : en effet, les blocs d'activation détectés par cette règle peuvent aussi l'être par la règle présentée dans la section 6.2.2. Puisque la règle détectant les fonctions génériques détecte un plus grand nombre de blocs dans la pile, on doit lui assigner la priorité la plus élevée pour s'assurer que tous les blocs représentant le motif sont bien traités.

Cet exemple concret laissant entrevoir les problèmes d'inter-dépendance et modularité pouvant apparaître dans la pratique lors de la mise au point d'un ensemble non trivial de règles de remplacement pour la prise en charge complète d'un langage. En réalité, il est très probable que des règles de remplacement partagent ou dupliquent des filtres de bloc étant donné la similitude de certains motifs de code produit par un compilateur. Il est donc important de fixer soigneusement la priorité d'une règle pour s'assurer que la construction de la vue virtuelle se déroule correctement.

6.2.4 Règles complexes : masquer l'implantation des exceptions

Le langage Bigloo fournit plusieurs constructions pour réagir aux exceptions pouvant se produire durant l'exécution. L'une de ces constructions est la forme `try`, dont le fonctionnement est illustré ci-dessous :

```
1  (module exception1 (main go))
2
3  (define (go args)
4     (try
5      (let ((x 10))
6         (print (cons x args)))
7      (lambda (escape proc err obj)
8         (foo err)
9         (escape #f))) )
10
11 (define (foo err)
12    (print "error: " err))
```

L'expression `try` se divise en deux parties : une s-expression à exécuter et une fonction anonyme servant de traitant d'exception. La s-expression (ligne 5 à 6 dans l'exemple) est exécutée dans un environnement « protégé ». Si une erreur se produit durant son évaluation, les calculs sont interrompus et l'exécution continue dans le corps de la fonction anonyme.

La sémantique de la forme `try` de Bigloo diffère du `try/catch/finally` fourni en natif par la JVM. La forme `try` est donc émulée par une succession d'appels de fonction. Les fonctions en attente dans la pile au moment où l'exécution atteint la ligne 6 sont présentées ci-dessous :

```
(bugloo) (info stack)
#0 (<anonymous:1041:exception1.scm:5> ::proc) in file exception1.scm:6
#1 (funcall0) in file exception1.scm
#2 (<exit:1205> ::proc) in file Ieee/control.scm:229
#3 (dynamic-wind ::proc ::proc ::proc) in file control.scm
#4 (<exit:1014:exception1.scm:4> ::proc ::proc ::obj) in file ...
#5 (go ::pair) in file exception1.scm:4
#6 (bigloo_main ::obj) in file exception1.scm
#7 (main ::(array of java.lang.String)) in file exception1.scm
```

Après inspection, on voit que le code protégé des lignes 5 à 6 est compilé en une fonction anonyme (bloc 0) et que les blocs 1 à 4 représentent les appels intermédiaires produits par le compilateur entre la fonction `go` et le code protégé. Ce motif de code est problématique. En effet, il ne faut pas que la règle de remplacement à mettre au point essaye de détecter le fonction qui a fait usage du `try` (dans notre cas le bloc 5), car il peut s'agir d'une fonction anonyme ou d'un fonction générique, qui ne peuvent être correctement masquées que par leur règle respective. De plus, il ne suffit pas de masquer les blocs 0 à 4, car le bloc 5 (la fonction `go`) ne contient pas la bonne information de ligne. La bonne manière de traiter ce motif consiste à masquer les blocs 0 à 4, attendre qu'une autre règle de remplacement soit appliquée pour donner le prochain bloc d'activation dans la pile virtuelle, puis d'y substituer les informations de ligne et de variables locales par celles récupérées préalablement du bloc 0. Cela peut être réalisé à l'aide d'une substitution retardée :

```
(80
 [(["BgL_zc3anonymousza3.*"] ("bigloo.proc") "Object" (? any))
  ("funcall0" () "bigloo.proc" (\ 1))
  (["BgL_zc3exitza3.*"] any any "bigloo.runtime.Ieee.control")
  ("BgL_dynamiczd2windzd2" any any "bigloo.runtime.Ieee.control")
  (["BgL_zc3exitza3.*"] any any "bigloo.runtime.Ieee.control")]
 mask
 rrp-bigloo-try)
```

Les deux premiers filtres de bloc recherchent un appel d'ordre supérieur d'arité zéro. Les trois filtres suivants détectent les trois appels de fonctions produits pour implanter la forme `try`. Toutes les fonctions détectées par ce motif sont masquées. La fonction `rrp-bigloo-try` fait office de substitution retardée :

```
(define (rrp-bigloo-try frames)
   (let ((saved-frame (car frames)))
      (lambda (f)
         (stackframe-line-set! f (delay (stackframe-line saved-frame)))
         f)))
```

La substitution retardée λ_σ reçoit en paramètre la liste des blocs d'activation détectés par le motif précédent (notés `frames`) et retourne une fermeture σ qui sera appliquée au prochain bloc d'activation qui apparaîtra dans la vue virtuelle (noté `f`). On voit que la création de σ provoque la capture du bloc 0 (noté `exc-frame`).

Dans notre exemple, au moment où le bloc 5 sera inséré dans la vue virtuelle, ses informations de ligne et de variables locales seront remplacées par celle du bloc 0. Il est intéressant de noter que durant sa création, la fonction anonyme représentée par le bloc 0 a capturé les variables locales définies dans la fonction `go`, il n'est donc pas nécessaire de fusionner la liste des variables du bloc 0 avec celle du bloc 5. Après substitution, la vue virtuelle est complètement nettoyée :

```
(bugloo) (info stack filter)
#0  (go ::pair) in file exception1.scm:6
```

Comme indiqué dans la figure 4.4, plusieurs substitutions σ_i peuvent être en attente jusqu'à ce que l'algorithme de construction de la vue virtuelle détermine un nouveau bloc d'activation à insérer dans la vue virtuelle. La définition de l'algorithme précise que ces substitutions sont appliquées dans l'ordre de leur création. Cette propriété donne l'assurance qu'une pile contenant plusieurs motifs de code `try` imbriqués sera correctement reconstruite dans la vue virtuelle.

6.3 Filtrage de l'exécution pas-à-pas

La section précédente a montré comment il était possible de masquer les détails de compilations qui apparaissent dans la pile suite à la compilation des programmes Bigloo. Cette section décrit les personnalisations supplémentaires à mettre en œuvre afin que l'exécution pas-à-pas ne soit pas perturbée par la présence de fonctions non désirables du point de vue du débogueur.

6.3.1 Filtrage simple

Durant l'exécution pas-à-pas, il faut filtrer les sauts amenant l'exécution dans la bibliothèque standard Bigloo, afin d'éviter à l'utilisateur de voir les détails d'implantation des fonctions standards (`map`, `list`, etc...) ou l'instanciation des boîtes enrobant certains types de données lors d'appels de fonctions (cf. chapitre 4.5, page 56). Pour cela, il suffit d'utiliser le support natif de filtrage de saut fournit par JVMTI (cf. 4.6.3, page 4.6.3) :

```
(map (lambda (x) (filter class add ,(format "bigloo.runtime.%s.*" x)))
     '("Jlib" "Lalr" "Llib" "Match" "Mlib" "Pp" "R5rs" "Read" "Rgc"))
```

Le filtre précédent est actif lorsqu'un saut arrive dans un package JVM contenant des classes de la bibliothèque standard Bigloo. On ne peut pas filter `bigloo.runtime.*` directement car il faut autoriser les sauts effectués dans l'évaluateur (c'est-à-dire les classes `bigloo.runtime.Eval.*`).

6.3.2 Filtrage par sauts virtuels

Le mécanisme de filtrage de sauts de JVMTI permet juste d'éviter de s'arrêter dans les fonctions de la bibliothèque standard Bigloo. Pour obtenir une exécution pas-à-pas exempte de toute perturbation, il est nécessaire de mettre au point plusieurs sauts virtuels, comme l'explique la section 4.5, page 56. La suite de cette section présente différents types de problèmes d'exécution pas-à-pas et les sauts virtuels permettant de les résoudre.

Filtrer les appels d'ordre supérieur

Les appels d'ordre supérieur sont implantés par des appels intermédiaires à des fonctions synthétiques produites dans les classes utilisateur. Il faut donc filtrer ces appels intermédiaires (cf. 6.2.2) à l'aide de sauts virtuels.

```
#0  (plus2 ::int) in file exemple1.scm:4
#1  (_plus2 ::proc ::obj) in file exemple1.scm
#2  (funcall1 ::obj) in file exemple1.scm
```

Lorsqu'un saut entrant s'arrête dans une fonction `funcall`n, il faut reprendre l'exécution jusqu'à la prochaine fonction. Il faut faire de même si cette fonction est un adaptateur servant à vérifier le typage.

Une difficulté posée par les saut virtuels est qu'ils doivent analyser les motifs de code dans le sens opposé à celui employé pour la construction de la vue virtuelle de pile (cf. 4.4, page 53) . Or, lorsque le saut virtuel doit être appliqué (c'est-à-dire dès qu'on entre dans une fonction `funcall`n) tous les blocs d'activation du motif de l'appel d'ordre supérieur ne sont pas encore présents sur la pile. Il faut donc concevoir plusieurs sauts virtuels pour traiter entièrement le motif. Voici un premier filtre pour masquer les sauts arrivant dans la fonction `funcall`n :

```
(bugloo) (step virtual add 100
            [(["(funcall([0-4])|apply)"] ("bigloo.proc" . re-args-ok)
                                         "Object" any)]
            NEXT)
```

Le filtre ci-dessus est actif lorsque la fonction en sommet de pile s'appelle `funcall`n ou `apply`. Le premier argument de la fonction doit être du type `proc`. La fonction `re-args-ok` vérifie que les derniers arguments sont au nombre de `n` et qu'ils sont du type `Object`.

La fonction `funcall`n peut appeler un adaptateur ou directement la fermeture utilisateur. Il faut pouvoir les différencier sans avoir la totalité du motif de l'appel d'ordre supérieur sur la pile. Pour cela, nous mettons au point un test plus coûteux que celui utilisé dans la construction de la vue virtuelle de pile :

```
(bugloo) (step virtual add 100
            [((? "(?:BgL)?_.*") ("bigloo.proc" . (? re-step-args))
                                "Object" (? re-stub))
             (["\\4"] re-step-args "Object" (\ 3))]
            NEXT)
```

Le saut virtuel précédent est activé si la fonction en sommet de pile représente un adaptateur ou une fermeture, c'est-à-dire si son nom commence par un tiret, si le type de son premier argument est un `proc` et en vérifiant que les arguments suivants sont des `Objects` (grâce à la fonction `re-step-args`). Dans ce cas, `re-stub` reçoit le nom de la classe dans laquelle est définie la fonction et s'assure que cette classe représente un module Bigloo. Puis `re-stub` inspecte cette classe à la recherche d'une fonction aux caractéristiques identiques et sans le tiret en début de nom. Si une telle fonction existe, la fonction en sommet de pile est un adaptateur. Le deuxième filtre de bloc doit alors détecter une fonction `funcall`n (le nom exact `\4` est fixé par `re-stub`) définit dans la même classe que la fonction précédente. Lorsque ce motif est détecté, l'exécution doit continuer jusqu'à la prochaine fonction.

L'action `NEXT` est utilisée pour les sauts virtuels, ce qui leur permet d'être applicable en cas de saut sortant. Lorsque l'exécution revient dans la fonction adaptateur, le second saut est appliqué. L'exécution continue automatiquement jusqu'à la fonction `funcall`, ce qui active le premier saut et provoque la sortie de cette fonction. Le saut sortant est donc correct du point de vue de l'utilisateur.

Saut complexe : sortir d'un motif de traitement d'exception

Pour certains motifs de code produits, il ne suffit pas d'éviter de s'arrêter dans les fonctions synthétiques pour obtenir une exécution pas-à-pas correcte. Prenons l'exemple de l'implantation de la forme `try` :

```
#0 (<anonymous:1041:exception1.scm:5> ::proc) in file exception1.scm:6
#1 (funcall0) in file exception1.scm
#2 (<exit:1205> ::proc) in file Ieee/control.scm:229
#3 (dynamic-wind ::proc ::proc ::proc) in file control.scm
#4 (<exit:1014:exception1.scm:4> ::proc ::proc ::obj) in file ...
#5 (go ::pair) in file exception1.scm:4
```

Dans ce motif, les blocs 0 et 5 représentent du code utilisateur. Supposons que nous ayons déjà mis au points quatre sauts virtuels avec une règle `NEXT` pour qu'un saut entrant effectué dans le bloc 5 saute automatiquement les blocs 4 à 1 et termine dans la fonction

anonyme du bloc 0. Si, à partir de ce point, on effectue un saut sortant, les sauts virtuels feront automatiquement revenir l'exécution jusqu'au bloc 5. Or, les blocs 0 et 5 représentent la même fonction dans le code source. Le saut sortant aurait donc un effet erroné du point de vue de l'utilisateur.

Pour que le saut sortant effectue bien l'opération logique escomptée par l'utilisateur, il est possible de spécialiser son comportement pour les blocs d'activation représentant des blocs de gestion d'exceptions Bigloo (cf. 6.2.4). Lorsqu'un saut sortant est effectué, le nouveau comportement demandera au débogueur l'exécution de six sauts sortants « natifs JVM » consécutifs. Cela provoquera la sortie des blocs d'activation représentant l'implantation de la forme `try` de Bigloo (blocs 0 à 4) ainsi que la fonction utilisateur courante (bloc 5). Un mécanisme similaire est utilisé pour sortir des fonctions génériques ou des fonctions de l'évaluateur Bigloo.

6.4 Filtrage avancé : masquer l'interprète Bigloo

Dans le langage Scheme, du code additionnel peut être chargé pendant l'exécution. Les nouvelles fonctions créées durant l'évaluation de code chargé dynamiquement sont compilées en une représentation intermédiaire *ad-hoc* et évaluées dans une pile annexe maintenue par l'interprète Bigloo. Une conséquence directe est que la pile native de la JVM n'est pas capable de visualiser ces fonctions interprétées, car elles ne sont pas compilées en code-octet JVM.

Cette section souligne encore l'utilité de la technique de construction de vue virtuelle, en présentant un ensemble de règles de remplacement avancées qui permettent la visualisation de deux représentations « disjointes » de code dans une unique vue unifiée de la pile. La suite de la section décrit brièvement le fonctionnement de l'interprète Scheme, puis présente les règles de remplacement adéquats, faisant usage de blocs d'activation virtuels, de priorités et de substitutions retardées.

6.4.1 Principe de fonctionnement de l'interprète Bigloo

L'interprète Bigloo utilise une représentation intermédiaire — que nous appellerons du code-octet — pour chaque construction de base du langage de programmation : la création de variable (`let`, `let*`, `letrec`), l'accès ou la mutation des données (`set!`), la séquence (`begin`), la conditionnelle (`if`), la création de fonction (`define`, `lambda`), l'appel de fonction et l'appel récursif terminal de fonction. L'évaluation d'un code-octet est effectuée au moyen d'une fonction de bibliothèque appelée `evmeaning`. Cette fonction implante un simple évaluateur symbolique qui gère son propre environnement, c'est-à-dire un ensemble de couples variable-valeur. Contrairement à d'autres interprètes de code-octet plus complexes, `evmeaning` se sert de la pile native JVM pour évaluer les appels de fonction. Pour comprendre comment sont encodées les expressions interprétées, considérons l'exemple suivant :

```
1 (define (go args)
2    (let ((f (eval '(lambda (x) (+ x 1)))))
3       (f 10)))
```

Dans cet exemple, `go` est une fonction compilé en code-octet JVM qui fait un appel à l'interprète pour créer une fonction interprétée. La fonction créée en retour est « enveloppée » dans une fermeture compilée en code-octet JVM qui provient de la bibliothèque

d'exécution de Bigloo. C'est cette fermeture qui est retournée par `eval` à la ligne 2. Par la suite, quand `f` est appelée à la ligne 3, le code contenu dans la fermeture fera un appel à `evmeaning` pour évaluer la fonction interprétée. Grâce à cet approche, il est possible d'appeler de manière transparente des fonctions interprétées depuis des fonctions compilées et *vice versa*.

Un bloc d'activation `evmeaning` dans la pile est responsable de l'évaluation d'une abstraction à la fois. En corollaire, l'évaluation d'une fonction entière peut introduire plusieurs blocs d'activation `evmeaning` consécutifs dans la pile. Considérons l'exemple ci-dessous :

```
1 (begin
2    (begin
3       (print 1)
4       (print 2))
5    (print 3))
```

L'évaluation du `begin` consiste à évaluer tous ses arguments en séquence. Ainsi, juste avant d'exécuter le `print` à la ligne 3, il y a deux blocs d'activation `evmeaning` dans la pile : un pour l'évaluation pendante du `begin` de la ligne 1 et un autre plus récent pour l'évaluation du `begin` imbriqué à la ligne 2. Il n'y aurait eu qu'un seul bloc d'activation `evmeaning` dans la pile si le `begin` imbriqué était la dernière expression du premier `begin`, car Scheme spécifie que l'évaluation des expressions situées en position récursive terminale ne doit pas consommer d'espace en pile.

À cause de leur représentation interne, les fonctions interprétées n'apparaissent pas dans la pile. À leur place, la pile contient des appels à la fonction d'évaluation `evmeaning`. L'exemple de la figure 6.2 permet de visualiser ce qui se passe dans la pile JVM durant leur évaluation.

Lorsque le programme compilé de la figure 6.2(a) est exécuté, il charge dynamiquement le fichier source de la figure 6.2(b), ce qui a pour effet de définir une fonction interprétée `foo1`. Ensuite, l'interprète est appelé pour évaluer un appel à la fonction `foo1` avec en paramètre la valeur `bar`. Supposons que nous avions précédemment posé un point d'arrêt interprété (cf. 6.1.2) dans `foo1` à la ligne 4. Au moment où le point d'arrêt est atteint, les fonctions suivantes sont dans la pile JVM :

```
1 (module example4
2    (main go))
3
4 (define (go args)
5    (load "test1.scm")
6    (eval '(foo1 "bar")))
```

(a) Le programme compilé

```
1 (define (foo1 o)
2    (let ((x 10))
3       (let ((y (* x 20)))
4          (car o)
5          (print "ok"))
6       (print (+ x o)) ))
```

(b) Le code chargé dynamiquement

FIG. 6.2: L'évaluation d'une fonction interprétée

```
(bugloo) (info stack)
#0  (breakpointReached ::int) in file Eval/evcompile.scm:622
#1  (check-breakpoint-reached ::obj) in file Eval/evcompile.scm:627
#2  (evmeaning ::obj ::obj ::obj) in file Eval/evmeaning.scm:143
#3  (evmeaning ::obj ::obj ::obj) in file Eval/evmeaning.scm:269
#4  (eval ::obj ::obj) in file Eval/eval.scm:124
#5  (go ::pair) in file example4.scm:6
#6  (bigloo_main ::obj) in file example4.scm
#7  (main ::(array of java.lang.String)) in file example4.scm
```

Les deux premiers blocs d'activation exhibent l'implantation des points d'arrêt interprétés. On peut masquer ces blocs avec la règle de remplacement suivante :

```
(100
 [("breakpointReached" any any (? "bigloo.runtime.Eval.evcompile"))
  ("BgL_checkzd2breakpointzd2reachedz00" any any (\ 1))]
 mask)
```

Le reste de pile montre bien le problème de débogage auquel l'utilisateur est confronté lorsqu'il utilise des fonctions interprétées : La pile ne présente que les fonctions compilées (blocs 7 à 4) et cache les fonctions interprétées dans l'implantation de l'évaluateur (blocs 3 à 2). Il est donc nécessaire de concevoir des règles de remplacements capables de remplacer les occurrences de la fonction `evmeaning` dans la pile par les fonctions interprétées en cours d'évaluation. La suite de la section décrit toutes ces règles de remplacements.

6.4.2 Remplacer les blocs d'activation de l'interprète dans la vue virtuelle

Dans la pile du programme de la figure 6.2, les deux blocs d'activation `evmeaning` représentent l'exécution de l'interprète : le bloc 3 est responsable de l'évaluation du code-octet `let` de la ligne 2 ; le bloc 2 représente l'évaluation du `let` emboîté de la ligne 3. Il n'y a pas de bloc d'activation `evmeaning` pour l'évaluation de l'appel à `foo1` car c'était un appel récursif terminal qui a entre temps disparu de la pile. En fait, un succession de blocs d'activation `evmeaning` consécutifs dans la pile représente toujours l'évaluation d'une seule fonction utilisateur. La règle de remplacement suivante prend en charge ce type de motif représentant l'interprète :

```
(70
 [(+ ("evmeaning" any any "bigloo.runtime.Eval.evmeaning"))
  (? (["BgL_zc3anonymousza3.*"] ("bigloo.proc" . (? re-bigloo-args2))
      any (? "bigloo.runtime.Eval.evmeaning"))
     (["\\2"] (\ 1) "Object" (\ 3)) )]
 sf-bigloo-evmeanings)
```

La règle montre qu'il est possible d'inspecter les structures de données synthétiques produites par le compilateur Bigloo pour retrouver de l'information logique qui n'est pas directement visible dans la pile. Une fois qu'une suite consécutive de blocs d'activation `evmeaning` a été détectée, elle est passée en argument à la fonction `sf-bigloo-evmeaning`, qui utilise les valeurs des variables locales du premier bloc d'activation pour reconstituer un bloc d'activation virtuel :

```
1  (define (sf-bigloo-evmeaning frames)
2     (let* ((em (car frames))
3            (bc (env-val (list-ref (frame-env em) 0)))
4            (names (env-val (list-ref (frame-env em) 1)))
5            (vals  (env-val (list-ref (frame-env em) 2))))
6        (list (create-virtual-frame
7               (fun-name-from-bytecode bc)
8               (fun-args-from-bytecode bc)
9               "java.lang.Object"
10              (fun-def-from-bytecode bc)
11              (fun-file-from-bytecode bc)
12              (fun-line-from-bytecode bc)
13              (make-env-tuples names vals)))))
```

Dans la fonction `evmeaning`, le premier argument représente le code-octet en cours d'évaluation (la variable `bc` à la ligne 3), à partir duquel on peut retrouver la vraie fonction utilisateur en cours d'évaluation. Le second et le troisième argument (lignes 4 et 5) représentent l'environnement de la fonction interprétée, respectivement les noms et les valeurs des variables capturées et locales[4]. Les valeurs des six premiers attributs du bloc d'activation virtuel (lignes 7 à 12) proviennent d'information extraites du code-octet courant `bc`. La liste des arguments et la liste des variables locales du bloc d'activation virtuel sont construites à l'aide du deuxième et du troisième argument de la fonction `evmeaning` (ligne 13). Après application de cette règle de remplacement, la vue virtuelle de la pile devient :

```
(bugloo) (info stack filter)
#0  (foo1 ::obj) in file test1.scm:4
#1  (eval ::obj ::obj) in file Eval/eval.scm:124
#2  (go ::pair) in file example4.scm:6
```

6.4.3 Règle complexe : Masquer les appels de fonctions interprétées

Filter les blocs d'activation responsable de l'appel

Dans une fonction Bigloo interprétée, tous les appels de fonctions sont des appels d'ordre supérieur. Ces appels sont implantés différemment de ceux présentés précédemment et doivent donc être traités par une règle de remplacement spécifique.

```
1  (module example5
2     (main go))
3
4  (define (go args)
5     (load "test2.scm")
6     (eval '(foo2 100)))
```

(a) Le programme compilé

```
1  (define (foo2 o)
2     (bar2 o)
3     o)
4
5  (define (bar2 o)
6     (+ o 2))
```

(b) Le code chargé dynamiquement

FIG. 6.3: L'évaluation d'une fonction interprétée

[4]Dans l'interprète Bigloo, les variables sont polymorphes ; en conséquence, l'interprète ne conserve pas d'information de type dans l'environnement.

Le programme présenté en figure 6.3 illustre cette différence d'implantation. Supposons que l'exécution du programme de la figure 6.3(a) soit suspendue après avoir atteint un point d'arrêt interprété à la ligne 6. À cet instant, l'état de la pile brute est le suivant :

```
(bugloo) (info stack)
#0  (breakpointReached ::int) in file Eval/evcompile.scm:622
#1  (check-breakpoint-reached ::obj) in file Eval/evcompile.scm:627
#2  (evmeaning ::obj ::obj ::obj) in file Eval/evmeaning.scm:143
#3  (<anonymous:1890> ::proc ::obj) in file Eval/evmeaning.scm:572
#4  (funcall1 ::obj) in file Eval/evmeaning.scm
#5  eval_funcall_1(bigloo.proc, java.lang.Object) in file foreign.java
#6  (evmeaning-funcall-1 ::obj ::obj ::obj) in file Eval/evmeaning.scm
#7  (evmeaning ::obj ::obj ::obj) in file Eval/evmeaning.scm:384
#8  (evmeaning ::obj ::obj ::obj) in file Eval/evmeaning.scm:269
#9  (eval ::obj ::obj) in file Eval/eval.scm:124
#10  (go ::pair) in file example5.scm:6
#11  (bigloo_main ::obj) in file example5.scm
#12  (main ::(array of java.lang.String)) in file example5.scm
```

Le bloc 2 représente la fonction `bar2`, les blocs 7 et 8 la fonction `foo2`. Les blocs intermédiaires de 3 à 6 exposent l'implantation de l'appel d'ordre supérieur. On peut voir que ce type d'appel de fonction interprétée n'est rien de plus qu'un appel d'ordre supérieur classique précédé de deux appels de fonctions spécifique à l'interprète Bigloo : le bloc 6 représente la fonction qui met en œuvre l'évaluation d'un appel de fonction interprété et le bloc 5 est une fonction intermédiaire qui déclenche un appel d'ordre supérieur classique. Ces deux blocs d'activation responsables de l'appel peuvent être masqués avec à la règle de remplacement suivante :

```
(70
   [
    (["eval_.*"] any any "bigloo.foreign")
    (["BgL_evmeaningzd2.*"] any any (\ 3))]
   mask)
```

Ces deux fonctions font partie de la bibliothèque d'exécution de Bigloo et sont uniquement utilisées dans le contexte de l'appel d'ordre supérieur. Il n'est donc pas nécessaire de vérifier l'arité de ces fonctions pour s'assurer qu'elles font partie du motif de code responsable de l'appel de fonction interprété. Cela entraîne un léger gain de temps.

Substitution retardée : masquer la préparation des appels de fonction

Il reste à prendre en charge l'évaluation des expressions passées en paramètre lors d'un appel de fonction : en effet, dans l'interprète Bigloo, les arguments d'un appels de fonction utilisateur sont évalués dans une fonction privée de la bibliothèque. Ce fonctionnement particulier est illustré dans la figure 6.4.

Considérons que l'exécution du programme de la figure 6.4(a) ait été suspendue après avoir atteint un point d'arrêt interprété dans la fonction interprétée `foo3` définie à la ligne 3 du listing 6.4(b). À ce moment précis, les fonctions suivantes sont dans la pile brute :

```
1 (module example6
2    (main go))
3
4 (define (go args)
5    (load "test3.scm")
6    (eval '(foo3 100)))
```

(a) Le programme compilé

```
1 (define (foo3 o)
2    (bar3 (let ((x 10))
3             (+ o x)))
4    o)
5
6 (define (bar3 o) (+ o 3))
```

(b) Le code chargé dynamiquement

FIG. 6.4: L'évaluation des arguments d'un appel de fonction interprétée

```
(bugloo) (info stack)
#0  (breakpointReached ::int) in file Eval/evcompile.scm:622
#1  (check-breakpoint-reached ::obj) in file Eval/evcompile.scm:627
#2  (evmeaning ::obj ::obj ::obj) in file Eval/evmeaning.scm:143
#3  (evmeaning-funcall-1 ::obj ::obj ::obj) in file Eval/evmeaning.scm
#4  (evmeaning ::obj ::obj ::obj) in file Eval/evmeaning.scm:384
#5  (evmeaning ::obj ::obj ::obj) in file Eval/evmeaning.scm:269
#6  (eval ::obj ::obj) in file Eval/eval.scm:124
#7  (go ::pair) in file example6.scm:6
#8  (bigloo_main ::obj) in file example6.scm
#9  (main ::(array of java.lang.String)) in file example6.scm
```

Pour mettre en œuvre l'évaluation de l'appel de fonction interprétée, `evmeaning` appelle la fonction de bibliothèque `evmeaning-funcall-1` (bloc 3). C'est seulement dans cette fonction que l'argument de l'appel d'ordre supérieur sera évalué, en appelant récursivement la fonction `evmeaning` (bloc 2). Ce bloc d'activation `evmeaning` « orphelin » doit être masqué dans la vue virtuelle car il ne représente pas l'évaluation d'une fonction logique définie par l'utilisateur. Néanmoins, les informations de ligne et de variables locales doivent être conservées car elles représentent la vraie position courante dans la fonction utilisateur en cours d'évaluation, ainsi que de l'environnement lexical « à jour » à cet endroit dans le source (c'est-à-dire contenant la variable `x`). La règle de remplacement à concevoir ne peut pas savoir comment traiter les blocs d'activation `evmeaning` 4 et 5 : en effet, ils pourraient représenter une fonction utilisateur, auquel cas il faudrait les conserver ; mais il pourraient aussi représenter la préparation d'un autre appel de fonction en attente, auquel cas ils faudrait les masquer.

Il est nécessaire d'utiliser une substitution retardée : l'idée est d'attendre qu'une règle virtualise les blocs 4 et 5 en un bloc d'activation virtuel représentant la fonction `foo3`, puis de le mettre à jour en y réinjectant les informations de ligne et de variables locales du bloc 2, ces informations ayant été préalablement sauvegardées dans une substitution. Ce comportement est mis en œuvre à l'aide de la règle de remplacement et de la substitution retardée suivantes :

```
(11
   [(+ ("evmeaning" any any "bigloo.runtime.Eval.evmeaning"))
    (["BgL_evmeaningzd2.*"] any any "bigloo.runtime.Eval.evmeaning")]
   mask
   rrp-bigloo-evmeaning-prep-call)
```

```
(define (rrp-bigloo-evmeaning-prep-call frames)
  (let ((ef (create-eval-stackframe (car frames))))
    (lambda (f)
      (stackframe-line-set! f (delay (stackframe-line ef)))
      (stackframe-localvars-set! f (delay (stackframe-localvars ef)))
      f)))
```

6.4.4 Masquer l'évaluation des macros dans l'interprète

Le langage Scheme propose un mécanisme de macro-expansion pour remplacer une forme syntaxique dans le source par un ensemble d'expressions calculées au moment de l'expansion. Les expressions macros-expansées à la compilation ne présentent pas de problème en ce qui concerne le débogage. Toutefois, la macro-expansion peut aussi se produire à l'exécution lorsque l'interprète Bigloo évalue des expressions. Il est donc nécessaire de masquer l'implantation de la macro-expansion.

Le système de macro de Bigloo est une mise en œuvre de la technique appelée *Expansion Passing Style* ou EPS [DFH86]. Dans ce schéma, les extensions syntaxiques sont manipulées par des fonctions appelées *expanders*. Un *expander* reçoit deux arguments : la forme syntaxique à expanser et un *expander* devant être appliqué au résultat de la macro-expansion du premier argument. Par récurrence, la macro-expansion retourne une s-expression composée uniquement de formes primitives du langage. Ce style d'expansion est similaire dans l'esprit au style de programmation CPS [GLS78].

```
1 (define-macro (add1 el)
2    `(+ ,el 1)
```

(a) macro utilisateur

$\rightsquigarrow$

```
1 (add-expander! 'add1
2   (lambda (x.%% e.%%)
3     (e.%% (let ((el (cadr x.%%))
4                 (.%% (cddr x.%%)))
5             (if (pair? .%%)
6                 (error "bad args")
7                 `(+ ,el 1)) )
8         e.%%))
```

(b) transformation en un expander équivalent

FIG. 6.5: L'implantation de la forme `define-macro` avec un expander

Dans sa quatrième révision, le langage Scheme fournit un système de macro différent à travers la forme `define-macro`. Dans Bigloo, ce système de macros est implanté en interne à l'aide d'*expanders*. À titre d'exemple, la figure 6.5 présente une macro utilisateur `add1` (6.5(a)) et l'*expander* équivalent (6.5(b)) construit par Bigloo pour implanter cette macro.

Contrairement aux implantations présentées précédemment dans ce chapitre, l'implantation de macro est singulière car le code produit par la macro-expansion est du code-octet Bigloo. Or, ce dernier a déja été pris en charge pour la construction de la vue virtuelle. Pour illustrer notre propos, considérons le listing de la figure 6.6. Supposons qu'un point d'arrêt interprété soit présent dans le fichier `file6.scm` à la ligne 2 (6.6(b)). Au moment où le programme compilé (6.6(b)) charge ce fichier, l'exécution se suspend dans la macro expansion de la fonction `foo4`.

La figure 6.7(a) présente la vue virtuelle de la pile au moment de la suspension. La

```
1 (module example7
2    (main go))
3
4 (define (go args)
5    (load "file6.scm"))
```

(a) Le programme compilé

```
1 (define-macro (add1 el)
2    `(+ ,el 1))
3
4 (define (foo4 x)
5    (add1 (* x x)))
```

(b) Le code chargé dynamiquement

FIG. 6.6: Macro-expansion durant l'évaluation d'un programme interprété

macro-expansion commence dans la bloc 8. Chaque appel à la fonction `initial-expander` est responsable de l'expansion d'une sous partie de la fonction `foo4`, jusqu'à l'appel de l'*expander* `add1`. On peut voir que l'émulation de cette macro entraîne l'apparition dans la pile de la signature de l'*expander*. De plus, les variables locales synthétiques produites par la compilation apparaissent dans le bloc d'activation.

Pour construire correctement la vue virtuelle de la pile d'exécution, il faut tout d'abord masquer les `expanders` de la bibliothèque d'exécution de Bigloo, car ils ne représentent pas d'intérêt particulier pour localiser la progression de la macro-expansion. Les règles de remplacement suivantes s'occupent du masquage :

```
(50
 [(+ (any any any ["^bigloo\\.runtime\\.Eval\\.(expd.*|expanders)$"]))]
 mask)

(50
 [(["BgL_zc3anonymousza3.*"] any any "bigloo.runtime.Eval.eval")
  (["funcall"] any any "bigloo.runtime.Eval.eval")]
 mask)
```

Ces deux règles masquent les fonctions provenant des fichiers nommés `expand.scm` et `expd*.scm`, ainsi que les fonctions anonymes du fichier `eval.scm`. Ce masquage est toutefois purement « cosmétique » et peut être débrayé si l'on souhaite déboguer la progression exacte de la macro expansion.

Pour construire un bloc d'activation virtuel correct à partir du bloc d'activation 0 qui représente la macro utilisateur, il faut modifier légèrement l'évaluateur Bigloo : lors de la création de fonctions anonymes utilisées comme macros (figure 6.5(b)), nous plaçons une marque dans le code-octet produit pour indiquer que la fonction est une macro, ainsi que le nombre d'argument que contient réellement cette macro utilisateur. Il suffit ensuite de modifier légèrement les fonctions `sf-bigloo-evmeaning` et `rrp-evmeaning-prep-call` pour créer un bloc d'activation virtuel représentant une macro ou une simple fonction interprétée selon le cas.

La figure 6.7(b) présente l'état de la vue virtuelle lorsqu'on applique les règles de remplacement et que l'on construit un bloc d'activation virtuel spécial. On voit que la signature originale de la macro utilisateur (bloc 0) est restaurée. De plus, les variables locales synthétiques produites générées par l'implantation de la forme `define-macro` (celles terminant par `.%%`) sont correctement masquées. Enfin, les blocs d'activation `initial-expander` sont conservés dans la vue virtuelle car ils peuvent être consultables pour localiser la partie exacte du code en cours de macro-expansion.

```
(bugloo) (info stack filter)
#0  (add1 ::obj :obj) in file file6.scm:2
#1  (<anonymous:1618> ::obj ::obj) in file Eval/eval.scm
#2  (initial-expander ::obj ::obj) in file Eval/expand.scm:76
#3  (<anonymous:1084> ::obj ::obj) in file Eval/expddefine.scm:82
#4  (expand-eval-external-define ::obj ::obj) in file Eval/expddefine.scm
#5  (expand-eval-define ::obj ::obj) in file Eval/expddefine.scm:117
#6  (<anonymous:2106> ::obj ::obj) in file Eval/expanders.scm:382
#7  (initial-expander ::obj ::obj) in file Eval/expand.scm:76
#8  (expand ::obj) in file Eval/expand.scm:51
#9  (eval ::obj ::obj) in file Eval/eval.scm:125
#10 (loadv ::obj ::obj) in file Eval/eval.scm:402
#11 (load ::obj) in file Eval/eval.scm:364
#12 (go ::pair) in file exemple6.scm:5
(bugloo) (info args filter)
el (::extended_pair) = (* x x)
.%% (::nil) = ()
x.%% (::extended_pair) = (add1 (* x x))
e.%% (::bigloo.runtime.Eval.expddefine) = procedure in expddefine.scm:80
```

(a) vue virtuelle sans filtrage de la forme `define-macro`

```
(bugloo) (info stack filter)
#0  (add1 ::obj) in file file6.scm:2
#1  (initial-expander ::obj ::obj) in file Eval/expand.scm
#2  (initial-expander ::obj ::obj) in file Eval/expand.scm:76
#3  (expand ::obj) in file Eval/expand.scm:51
#4  (eval ::obj ::obj) in file Eval/eval.scm:125
#5  (loadv ::obj ::obj) in file Eval/eval.scm:402
#6  (load ::obj) in file Eval/eval.scm:364
#7  (go ::pair) in file exemple6.scm:5
(bugloo) (info args filter)
el (::extended_pair) = (* x x)
```

(b) vue virtuelle tenant compte de la forme `define-macro`

FIG. 6.7: Macro-expansion durant l'évaluation d'un programme interprété

6.5 Rapport d'expérience sur l'implantation

6.5.1 Les blocs d'activation virtuels de Bigloo

Comme l'explique la section 4.6.1, la prise en charge du langage Bigloo nécessite de surcharger la classe BUGLOO `stackframe` pour représenter les nouveaux types de blocs d'activation. Une fonction anonyme ou une fermeture est ainsi représentée par une nouvelle classe `closure-stackframe`. Les accesseurs de cette classe utilisent JVMTI pour inspecter parcourir la liste des variables capturés conservées dans l'argument de type `proc` (l'environnement) de la fonction JVM originale. Ces objets sont ensuite « enrobés » dans un nouveau type de variable locale `captured-localvar` et comptabilisés parmi la liste des variables locales de la fonction.

L'implantation de la forme `try` est masquée par une substitution, qui crée un objet de type `try-stackframe`. Nous avons besoin d'une nouvelle classe pour pouvoir surcharger l'implantation par défaut du saut sortant (cf. chapitre 4.6.1, page 58) afin que la sémantique originale de ce saut soit conservée en présence de ce type de motif. Le bloc d'activation virtuel `generic-stackframe` représentant les fonction génériques fournit le même type de surcharge.

6.5.2 Masquer l'évaluateur

Le masquage de l'interprète est plus conséquente et demande de rajouter du code à la fois dans le débogueur et dans le débogué. Un bloc d'activation de l'interprète est représenté par la classe `eval-stackframe`. Ce type de bloc est construit à partir d'une référence sur un objet du débogué contenant le code octet en cours d'évaluation. Pour retrouver toutes les informations concernant la fonction en cours d'évaluation, il faut traverser le code-octet. La construction fait des appels distants dans la JVM du débogué pour effectuer la traversée directement dans la JVM du débogué. Cela permet de conserver des bonnes performances durant la construction de la vue virtuelle.

Des fonctionnalités de contrôle de l'exécution ont dû être rajoutées à l'interprète Bigloo pour supporter la pose de points d'arrêt fonctionnant sur du code octet Bigloo. En plus des points d'arrêts interprétés, présentés dans la section 6.1.2, l'interprète a dû être légèrement modifié pour permettre une exécution pas-à-pas dans du code interprété à la manière des blocs d'activation virtuels `try-stackframe`. Un saut se traduit par un appel à une fonction spéciale de l'interprète dans la JVM du débogué. Un points d'arrêt JVM est placé dans cette fonction pour suspendre le débogué. Lorsque ce point d'arrêt est atteint, le débogueur signale la fin d'un saut pour masquer la vrai nature de la suspension.

6.5.3 Présence des informations de débogage

La vue logique de la pile d'exécution est construite en regroupant des informations disséminées dans la pile JVM. En corollaire, il n'est pas possible de reconstruire la vue s'il ne reste plus suffisamment d'information logique après la phase de compilation. Dans le cas de Bigloo, il a fallu adapter légèrement les motifs de code produit par le compilateur pour rajouter quelques informations de débogage.

La macro-fonction `define-method` produit une fonction qui perd les information de typage pour ses arguments (cf. section 6.2.3). Il a donc fallu modifier cette macro pour que le nom de la fonction produite inclue le type de l'argument générique. Cet encodage *ad-hoc* de l'information de débogage permet de reconstruire la fonction utilisateur originale. De plus, avec cet encodage, la construction de la vue reste effective même lorsque le module contenant la fonction générique n'est pas compilé en mode débogage.

Lors de la compilation d'un fonction anonyme, la valeur des variables capturées est conservée dans un tableau de l'objet `proc` (cf. section 6.2.2). Une variable est représentée par un indice dans ce tableau, mais son nom n'est pas conservé. Il a fallu modifier légèrement la code produit par la compilation des fermetures pour que le premier argument de la fonction (l'objet `proc`) ait un nom spécial qui encode le nom des variables capturées. Par exemple, l'argument sera nommé `<env:3_4:foobarz>` si la fonction anonyme capture deux variables `foo` et `barz` de longueur respectives 3 et 4. Comme ce ne sont pas des variables locales natives JVM, elle doivent être représentées dans Bugloo par une nouvelle classe `captured-localvar`.

6.5.4 Adaptations apportées à l'interprète Bigloo

Dans l'interprète Bigloo original, seuls les informations de lignes étaient conservées après la compilation d'un s-expression en code-octet. La fonction `evmeaning` prenait deux arguments : le code-octet à évaluer et l'environnement courant, représenté une liste de valeurs. Dans l'environnement, chaque variable locale était représentée par une position particulière dans la liste. Dans le débogueur, ce type de variable interprétée est représenté par une classe `eval-localvar`. Il a fallu modifier légèrement la fonction `evmeaning` pour lui adjoindre un troisième argument servant à conserver le nom de chaque variable locale de l'environnement courant. De même, il a fallu changer l'implantation des fonctions en charge de mettre à jour l'environnement local, comme par exemple l'évaluation du code-octet `let`.

Lorsque l'interprète Bigloo crée une nouvelle fonction, il pose une marque de débogage dans le tout premier code-octet du corps de la fonction afin de se souvenir du nom utilisateur de cette fonction. Une fois que la fonction est en cours d'évaluation, la fonction `evmeaning` n'a plus accès au sommet de la fonction interprété. Pour conserver les informations de débogage, l'interprète a été modifié pour propager l'information de débogage dans tout le corps des fonctions qu'il vient de crée. Cela est nécessaire pour construire correctement la vue virtuelle de la pile.

6.6 Conclusion

Ce chapitre a présenté un ensemble de fonctionnalités supplémentaires développés pour tenir compte des spécificités du langage fonctionnel Bigloo. Il a décrit un ensemble complet de règles de filtrage développées pour masquer la compilation délicate du langage Bigloo. Une série de règles de remplacement permet de construire une vue virtuelle de la pile d'exécution sans laisser transparaître de détails de compilation. Ensuite, un ensemble de sauts virtuels permet de contrôler l'exécution pas-à-pas afin qu'elle ne soit plus perturbée par la compilation. Enfin, un soin particulier a été apporté au masquage de l'interprète embarqué pour permettre d'afficher dans une seul vue unifiée la pile JVM et la pile de l'interprète.

Les différents exemples exposés dans ce chapitre ont montré qu'un grand nombre de motifs de compilation a pu être masqué sans aucune intervention dans le compilateur Bigloo ou dans sa bibliothèque d'exécution. D'autres motifs ont nécessité quelques légères modifications, comme l'ajout d'information de débogage dans les fermetures ou dans l'interprète Scheme embarqué. Après cette petite participation de la part des implanteurs de Bigloo, les différents motifs de code ont tous pu être pris en charge par les mécanismes de représentations virtuelles développés dans le débogueur. À titre indicatif, les règles de remplacement développées ainsi que les classes surchargées sont présentées en annexe.

En conclusion, ce chapitre a montré que grâce à l'API de programmation du débogueur et aux algorithmes de construction de vue virtuelle, les implanteurs de langages peuvent réaliser un support de débogage complet pour leur langage de haut niveau en peu de temps. Les différentes expérimentations réalisées montrent que les résultats obtenus grâce à ces mécanismes sont conformes aux exigences énoncées en introduction de ce manuscrit. En ce sens, ces expérimentations ont validé les mécanismes de représentations virtuelles de Bugloo.

Chapitre 7

Expérimentations sur d'autres langages

Le chapitre précédent a décrit de manière exhaustive le support de débogage développé pour le langage Bigloo. En particulier, il a présenté différents cas d'utilisation des règles de remplacement. Ce chapitre a pour but de montrer que ces techniques de représentations virtuelles sont génériques et peuvent être employées sur d'autres langages de haut niveau, aux caractéristiques et à la compilation bien différentes. Dans ce chapitre, il est présenté des extensions réalisées pour masquer les détails de compilation de deux implantations de langage : Rhino [Fou98], le compilateur ECMAScript [ECM99] de la Fondation Mozilla et Jython [Hug97], l'implantation du langage Python [Ros00] pour la JVM. Ce chapitre s'étend aussi sur une propriété fondamentale des techniques de représentation virtuelles développées dans Bugloo : permettre le débogage des programmes multi-langages de manière transparente pour l'utilisateur.

Ainsi, ce chapitre décrit les deux supports de débogage réalisés pour les implantations des langages Rhino et Jython. Puis, il décrit un exemple concret de débogage de programme multi-langage mêlant Scheme, ECMAScript et Python. Enfin, le chapitre termine en parlant de l'expérience acquise avec les règles de remplacement après le développement des trois différents supports de débogage.

7.1 Extension de débogage pour l'environnement Rhino

Le langage ECMAScript est devenu populaire avec l'apparition des *browsers* Web. C'est un langage de haut niveau assez « simple » et aux caractéristiques assez similaires aux langages fonctionnels à typage dynamique comme Scheme : il propose des fonctions anonymes, la capture de variable et dispose d'un évaluateur. Ce langage fournit un système d'objets à prototypes reposant sur les fonctions.

Rhino est une implantation du langage ECMAScript entièrement écrit en Java et qui peut être facilement interfacé avec Java. Développé à l'origine pour être un simple interprète d'ECMAScript, l'environnement Rhino a évolué pour fournir un compilateur ECMAScript vers code-octet JVM.

Cette section présente le support de débogage réalisé pour le compilateur Rhino. Elle décrit tout d'abord les règles de remplacements mise au point pour masquer les détails de compilation qui apparaissent dans la pile, puis elle présente la manière d'afficher les

variables et les objets du programme.

7.1.1 Prise en charge de la pile d'exécution

Le compilateur Rhino emploie un schéma de compilation qui présente des similitudes avec celui de Bigloo : il produit directement du code-octet JVM à partir de fichiers source ECMAScript. Chaque fichier source est compilé en une classe JVM. Les fonctions utilisateurs sont compilées en des fonctions statiques JVM. Le compilateur produit des tables de débogage standards dans les classes JVM pour conserver les informations de lignes. Cela permet par exemple à des débogueurs classiques d'opérer une exécution pas-à-pas correcte dans le code utilisateur.

Le langage ECMAScript offre des fonctions d'ordre supérieur et des fonctions anonymes. Le compilateur Rhino doit donc produire des fonctions intermédiaires pour implanter ces fonctionnalités dans la JVM. De la même manière que Bigloo, il a recours à un objet spécial nommé `NativeFunction` pour représenter les fermetures et utilise une fonction de dispersion (cf chapitre 6.2.2) pour effectuer les appels. Toutefois, contrairement à Bigloo, du fait de la nature des appels de fonctions dans ECMAScript [1], le compilateur Rhino n'utilise pas les variables locales natives de la JVM pour représenter les arguments des fonctions utilisateur ou leurs variables locales. Au lieu de cela, il s'appuie sur une gestion de son propre type de bloc d'activation *ad-hoc* durant l'exécution.

```
(bugloo) (info args)
#0 _c3(test, Context, Scriptable, Scriptable, Object[]) in file test.js:10
#1 call(Context, Scriptable, Scriptable, Object[]) in file test.js
#2 callName(Object[], String, Context, Scriptable) in file OptRuntime.java
```

(a) Motif de code produit par le compilateur

```
(100
 ((["^_c[0-9]+$"] ((? any) . (? any)) "java.lang.Object" (\ 1))
  ("call" (\ 2) "java.lang.Object" (\ 1))
  (? (["^call.*"] any any "org.mozilla.javascript.optimizer.OptRuntime")))
 sf-rhino-function)
```

(b) Règle de remplacement masquant le motif précédent

FIG. 7.1: L'appel par nom dans la plateforme d'exécution Rhino

La figure 7.1(a) présente le motif de code produit par le compilateur Rhino pour un appel de fonction provenant d'un fichier `test.js`. L'implantation de l'appel réside dans une fonction de bibliothèque appelée `callName` (bloc 2). Le premier argument est un tableau contenant tous les arguments de l'appel ECMAScript utilisateur. Le second argument est le nom de la fonction utilisateur à appeler. Dans la terminologie ECMAScript, le troisième argument est le *contexte* d'évaluation courant, il représente le bloc d'activation Rhino courant. Le dernier argument représente la chaîne de *portée* courante, c'est-à-dire l'ensemble des variables locales accessibles à cet endroit du programme. Le rôle de la fonction de bibliothèque est de rechercher dans la liste des fonctions définies à l'exécution l'objet `NativeFunction` adéquat et d'appeler sa fonction de dispersion (bloc 1). Dans la fonction de dispersion, le

[1] Notamment la sémantique des fonctions d'arité variable.

troisième argument représente le l'activation courante, un équivalent du pointeur `this` en ECMAScript.

Lorsque la fonction de dispersion appelle la fonction utilisateur correspondante, cette fonction reçoit en paramètre l'objet `NativeFunction` ainsi que tous les arguments reçus par la fonction de dispersion. Une fonction utilisateur compilée par Rhino est appelée _c*n*, selon son ordre d'apparition dans le fichier source. La règle de remplacement présentée dans la figure 7.1(b) permet de construire un bloc d'activation virtuel masquant ce motif de code synthétique. Le motif de bloc vérifie le nom de la fonction utilisateur et mémorise le type de son premier argument, ainsi que la liste des arguments restant. Puis, il s'assure que la fonction suivante s'appelle `call`, qu'elle attend les bons arguments et qu'elle est définie dans la même classe que la fonction précédente. Enfin le filtre vérifie que la troisième fonction du motif représente bien l'implantation d'un appel de fonction.

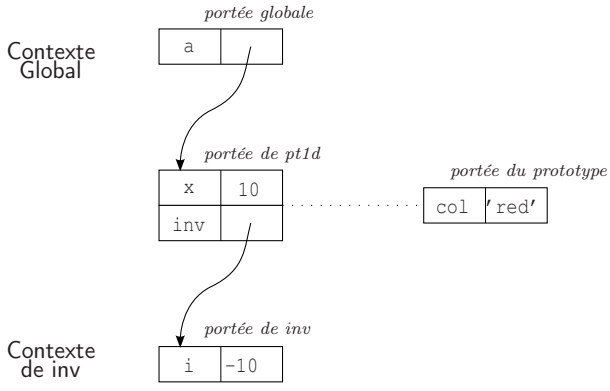
7.1.2 L'inspection des variables locales

Le langage ECMAScript fournit différents niveaux de variables locales, comme par exemple les variables capturées ou les variables accédées à l'aide du mot-clef `with`. Ces niveaux sont appelées *portées* des identificateurs. En ECMAScript, chaque portée est implantée par une sorte de table de hachage associant des variables à des valeurs. De plus, le programme peut modifier le contenu des ces portées dynamiquement, par exemple en ajoutant ou en retirant des variables locales.

Il existe différents types de portées. Par exemple, les objets structurés du langage sont des tables de hachage dans lesquelles on peut ajouter ou retrancher des champs. Dans une fonction, le mot-clef `with` permet d'évaluer du code en utilisant comme portée locale les champs d'un objet. Par ailleurs, en ECMAScript, les objets sont des *prototypes*. Les fonctions peuvent être « instanciées » afin de créer des objets dont les champs sont les variables locales. Lorsqu'un objet est instancié, il devient lui-même un nouveau prototype : ses champs peuvent être modifiés sans modifier le prototype de l'objet original. En revanche, si un champ est rajouté au prototype initial, tous les objets créés à partir de ce prototype profiteront automatiquement du champ supplémentaire. Le prototype définit donc un niveau particulier de portée.

```
1  function pt1d(x) {
2    this.x=x;
3    this.inv=function() {
4      var i=-x;
5      return i;
6    }
7  }
8  pt.prototype.color='red';
9
10 a=new pt1d(10);
11 print a.inv();
```

(a) Programme ECMAScript



(b) Niveaux lexicaux accessibles depuis `inv`

FIG. 7.2: Les différentes portées lexicales d'un programme ECMAScript

La plateforme Rhino maintient ses propres blocs d'activation ainsi que ses propres ni-

veaux de portée. Ainsi, lorsque l'utilisateur demande la liste des variables locales d'un bloc d'activation virtuel Rhino, le débogueur doit afficher toutes les variables définies dans les portées accessibles. La figure 7.2(a) illustre ce cas de figure. Le programme présenté crée un objet `pt1d` contenant un champ et une fonction. Puis il rajoute à son prototype un nouveau champ. Lorsque l'exécution est suspendue à la ligne 5, la liste des portées accessibles est présentée dans la figure 7.2(b).

Pour retrouver la liste des variables locales, il faut accéder à la portée de la fonction courante. Pour que l'inspection fonctionne même sans information de débogage, le débogueur n'utilise pas les arguments de la fonction *_cn*. Il retrouve la portée courante en interrogeant l'objet `NativeFunction` obtenu par le pointeur `this` de la fonction `call` (bloc 1 de la figure 7.1(a)).

Toutes les valeurs Rhino héritent de la classe primaire `Scriptable`. Pour leur affichage, l'extension de débogage Rhino appelle dans la JVM du débogué la fonction Rhino `toString` qui s'occupe du formatage des valeurs.

7.1.3 Conclusion de la prise en charge du langage

Le compilateur Rhino produit des motifs de code en utilisant les mêmes techniques (dispersion et captures de variables) que le compilateur Bigloo. Toutefois, l'implantation est différentes : le compilateur ne cherche pas à produire des noms de fonctions « intelligibles » et doit produire beaucoup plus de structures intermédiaires du fait de la nature fortement dynamique du langage. Malgré cela, les règles de remplacement peuvent masquer ces artefacts sans modifier le compilateur Rhino.

7.2 Extension de débogage pour l'environnement Jython

La compilation des programmes Jython diffère assez largement des deux autres implantations de langages évoquées jusqu'à présent. En effet, l'environnement Jython est à la base un interprète de Python pour la JVM. Dans cet environnement, le code utilisateur est compilé en un code-octet Jython intermédiaire, puis exécuté dans une machine à registres *ad-hoc*. L'interprète gère sa propre liste de blocs d'activation Python.

L'environnement Jython fournit un compilateur nommé `jythonc`. C'est une sorte d'évaluateur partiel dont le but est de produire du code « prédigéré » s'exécutant plus rapidement. Au lieu de produire directement du code JVM optimisé, `jythonc` produit du code Java représentant l'évaluation directe du code-octet Jython original.

7.2.1 Gestion de la pile d'exécution Jython

La figure 7.3 présente le résultat de la compilation d'un fichier source Python servant à construire une chaîne de caractères de manière itérative. À l'image de Bigloo et de Rhino, on voit que le compilateur a produit une fonction de dispersion `call_function` afin d'implanter les appels de fonctions d'ordre supérieur. Lorsque la fonction utilisateur `CounterString` a été compilée, son nom a été encodé pour marquer son arité. De plus, on voit que l'argument utilisateur a disparu de la signature de la fonction. À la place, toutes les fonctions produites par `jythonc` reçoivent un argument `PyFrame` qui encode un bloc d'activation Jython. Le corps de la fonction utilisateur utilise des emplacements mémoire dans ces blocs d'activation (fonction `setlocal`) pour créer ou modifier des variables locales *ad-hoc*.

```
def CounterString(length):
    buffer = ""
    for i in range(START, length):
        buffer = buffer + str(i) + SPACER
    return buffer

print CounterString(10)
```

(a) Fichier Python `test2.py`

```
public class test2 extends Object {
    public static class _PyInner extends PyFunctionTable {
        {...}
        private static PyCode c$0_CounterString=
          Py.newCode(2, new String[] {"length", "buffer", "i"},
                     "/home/dciabrin/code/python/test2.py", "CounterString",
                     false, false, funcTable, 0, null, null, 0, 1);

        public PyObject call_function(int index, PyFrame frame) {
            switch (index){
                case 0:
                return _PyInner.CounterString$1(frame);
                case 1:
                return _PyInner.main$2(frame);
                default:
                return null;
            }
        }

        private static PyObject CounterString$1(PyFrame frame) {
            //TemporaryVariables
            int t$0$int;
            PyObject t$0$PyObject, t$1$PyObject;

            //Code
            frame.setlocal(2, s$2);
            t$0$int = 0;
            t$1$PyObject = frame.getglobal("range").
                        __call__(frame.getglobal("START"), frame.getlocal(0));
            while ((t$0$PyObject = t$1$PyObject.__finditem__(t$0$int++))
                                != null) {
                frame.setlocal(2, t$0$PyObject);
                frame.setlocal(1, frame.getlocal(1)._add(frame.getglobal("str")
                .__call__(frame.getlocal(2)))._add(frame.getglobal("SPACER")));
            }
            return frame.getlocal(2);
        }

        {...}
        }
}
```

(b) Code Java produit par le compilation `jythonc`

FIG. 7.3: Exemple de code Python compilé

Hormis les noms de fonctions, toutes les informations de débogage des fonctions synthétiques produites par le compilateur sont conservées dans des structures de données *ad-hoc* de type `PyCode`. On y trouve en particulier le nom original de la fonction utilisateur, son arité, le nom du fichier source original ou encore le nom des variables locales et leur position dans les emplacements mémoire du bloc d'activation.

```
(bugloo) (info args)
#0  CounterString$1(PyFrame) in file test2.java:51
#1  call_function(int, PyFrame) in file test2.java:37
#2  call(PyFrame, PyObject) in file PyTableCode.java:208
#3  call(PyObject, PyObject, PyObject[], PyObject) in file PyTableCode.java
#4  __call__(PyObject) in file PyFunction.java:172
```

(a) Le motif de code produit pour l'appel de fonction

```
(100
 ((["^.*$([0-9]+)"] ("org.python.core.PyFrame") any (\ 1))
  ("call_function" ("int" "org.python.core.PyFrame") any (\ 1))
  (* (["call"] any any "org.python.core.PyTableCode"))
  ("__call__ sf-rhino-arity any "org.python.core.PyTableCode"))
 sf-rhino-function)
```

(b) Règle de remplacement pour le motif précédent

FIG. 7.4: Filtrage de l'appel de fonction Jython

La figure 7.4(a) décrit un appel de fonction d'arité 1 dans la plateforme Jython. Les fonctions sont représentées par des objets `PyFunction`. Dans, le bloc 4 du motif, la fonction `__call__` implante l'appel de fonction et reçoit l'argument de l'appel utilisateur dans un objet `PyObject`. Cette fonction appelle la fonction `call` dans le bloc 3. Le premier argument de cette fonction contient l'argument utilisateur. Le second argument contient la table d'association des variables globales. Le troisième argument contient la liste des valeurs par défaut pour cet appel de fonction. Enfin, le dernier argument correspond à la liste des variables capturées au cas où la fonction à appeler soit une fonction interne. La fonction `call` du bloc 3 affecte le premier emplacement mémoires du bloc d'activation avec la valeurs de l'argument utilisateur, puis appelle la fonction `call` du bloc 2. Le rôle de cette dernière est de rentrer dans un bloc d'exception puis d'appeler la fonction de dispersion `call_function`, qui elle même appellera la fonction utilisateur.

Pour masquer les appels de fonctions synthétiques produits par le compilateur Jython, on emploie la règle de remplacement de la figure 7.4(b). Son motif cherche une fonction utilisateur et conserve son arité. Le motif vérifie que la fonction utilisateur est suivie de sa fonction de dispersion et de `call`. Enfin, le motif s'assure que la dernière fonction du motif est bien un `__call__` qui contient autant d'argument que l'arité de la fonction utilisateur.

7.2.2 Gestion des variables de Jython

Pour afficher la liste des variables locales d'une fonction Jython, le débogueur doit inspecter les informations de débogage *ad-hoc* produites par `jythonc` dans le descripteur de la fonction (l'objet `PyCode` évoqué précédemment). Cela lui permet de retrouver la liste des noms de variables. Il doit ensuite retrouver leurs valeur qui sont conservées dans les em-

placements mémoires correspondant dans le bloc d'activation. Enfin le débogueur parcours la liste des variables capturées, au cas où la fonction soit une fonction interne. Même si le module Jython n'est pas compilé en mode débogage, l'objet `PyCode` est présent durant l'exécution. La construction de variables virtuelles est donc un mécanisme « sûr ».

L'affichage d'un objet est pris en charge par le support de débogage Jython si cet objet hérite de la classe `PyScriptable`. La valeur de l'objet est obtenue en appelant sa méthode JVM `toString` dans la JVM du débogué.

Enfin, le support de débogage permet de construire des objets virtuels à partir des objets qui héritent de `PyScriptable`, afin de masquer les champs synthétiques des structures Jython qui n'existent pas dans le fichier source. Pour retrouver les champs logiques d'une structure Jython, le débogueur appelle sa méthode `dir`, une méthode Python standard permettant de faire de l'introspection. En Python, l'introspection est une fonctionnalité qui peut être surchargée dans les classes utilisateurs. Utiliser la méthode `dir` permet de respecter les conventions d'introspection du programme utilisateur.

7.3 Exemple de débogage d'un programme multi-langages

Cette section illustre la manière dont les règles de remplacement peuvent être employées pour permettre de déboguer des programmes composés de plusieurs langages de haut niveau, là ou les débogueurs classiques ne sont plus d'aucune utilité.

```
1 function foo() {
2     Packages.multiscm.DYNAMIC_LOAD_INIT();
3     var f=Packages.multiscm.hello;
4     var c=Packages.multipy.code();
5     c.gee(f);
6 }
7
8 foo();
```

(a) fichier `multijs.js` (point d'entrée)

```
1 (module multiscm
2    (export hello))
3
4 (define hello
5    (lambda ()
6       (print "Hello Scheme!")))
7
8
```

(b) fichier `multiscm.scm`

```
1 from java.lang import Object;
2 class code(Object):
3     def gee(self, fun):
4         "@sig public java.lang.Object gee(bigloo.procedure fun)"
5         return fun.funcall0()
```

(c) fichier `multipy.py`

FIG. 7.5: Programme composé de trois fichiers source écrits dans différents langages : 7.5(a) ECMAScript ; 7.5(b) Bigloo ; 7.5(c) Jython.

L'exemple de la figure 7.5 présente le code source d'une programme utilisant trois langages sources différents : Rhino, Jython et Bigloo.

Le point d'entrée du programme à déboguer est l'expression située à la ligne 8 de la figure 7.5(a) effectuant un appel à la fonction Rhino `foo`. Cette fonction initialise tout d'abord la bibliothèque d'exécution Bigloo. Ensuite, elle crée une variable `f` pour référencer la fermeture Scheme contenue dans la variable `hello` du module `multiscm`. Enfin, elle crée une instance de la classe Python `code` et appelle sa méthode `gee` en lui passant en paramètre

la fermeture Scheme. Supposons qu'un point d'arrêt ait été posé dans le fichier `multiscm.scm` à la ligne 6. Le contenu de la pile au moment où l'exécution atteint cette ligne est présenté dans la figure 7.6. Manifestement, la pile contient beaucoup trop de fonctions synthétiques et de fonctions de bibliothèque pour que le programme puisse être débogué correctement. L'utilisation de règle de remplacement permet de remédier à ce problème. Le chapitre 6 a déjà décrit les règles nécessaires au masquage des motifs de code produits par le compilateur Bigloo. Il est maintenant nécessaire de concevoir un ensemble de règles spécifiques aux langages Rhino et Jython afin d'obtenir une vue virtuelle correcte de la pile du programme. Ces règles sont présentés dans les sections suivantes.

```
#0   (<anonymous:1003:multiscm.scm:4> ::proc) in file multiscm.scm:6
#1   (funcall0) in file multiscm.scm
#2   invoke0(reflect.Method, Object, Object[]) in file NativeMethodAccessorImpl.java
#3   invoke(Object, Object[]) in file sun/reflect/NativeMethodAccessorImpl.java:39
#4   invoke(Object, Object[]) in file sun/reflect/DelegatingMethodAccessorImpl.java:25
#5   invoke(Object, Object[]) in file java/lang/reflect/Method.java:585
#6   __call__(PyObject, PyObject[], String[]) in file PyReflectedFunction.java:160
#7   __call__(PyObject[], String[]) in file org/python/core/PyMethod.java:96
#8   __call__() in file org/python/core/PyObject.java:258
#9   invoke(String) in file org/python/core/PyInstance.java:244
#10  gee$1(PyFrame) in file multipy.java:45
#11  call_function(int, PyFrame) in file multipy.java:33
#12  call(PyFrame, PyObject) in file org/python/core/PyTableCode.java:208
#13  call(PyObject[], String[], PyObject, PyObject[], PyObject) in file PyTableCode.java:
#14  call(PyObject, PyObject[], String[], PyObject, PyObject[], PyObject) in file PyTable.
#15  __call__(PyObject, PyObject[], String[]) in file org/python/core/PyFunction.java:190
#16  __call__(PyObject[], String[]) in file org/python/core/PyMethod.java:96
#17  __call__(PyObject[]) in file org/python/core/PyObject.java:248
#18  _jcallexc(Object[]) in file org/python/core/PyObject.java:1958
#19  _jcall(Object[]) in file org/python/core/PyObject.java:1990
#20  gee(bigloo.proc) in file multipy.java:69
#21  invoke0(reflect.Method, Object, Object[]) in file sun/reflect/NativeMethodAccessor...
#22  invoke(Object, Object[]) in file sun/reflect/NativeMethodAccessorImpl.java:39
#23  invoke(Object, Object[]) in file sun/reflect/DelegatingMethodAccessorImpl.java:25
#24  invoke(Object, Object[]) in file java/lang/reflect/Method.java:585
#25  invoke(Object, Object[]) in file org/mozilla/javascript/MemberBox.java:174
#26  call(Context, Scriptable, Scriptable, Object[]) in file NativeJavaMethod.java:197
#27  call1(Function, Scriptable, Object, Context, Scriptable) in file OptRuntime.java:63
#28  _c1(multijs, Context, Scriptable, Scriptable, Object[]) in class multijs:5
#29  call(Context, Scriptable, Scriptable, Object[]) in class multijs (no line info...)
#30  callName0(String, Context, Scriptable) in file OptRuntime.java:105
#31  _c0(multijs, Context, Scriptable, Scriptable, Object[]) in class multijs:8
#32  call(Context, Scriptable, Scriptable, Object[]) in class multijs (no line info...)
#33  doTopCall(Callable, Context, Scriptable, Scriptable, Object[]) in file ContextFactory.
#34  doTopCall(Callable, Context, Scriptable, Scriptable, Object[]) in file ScriptRuntime.
#35  call(Context, Scriptable, Scriptable, Object[]) in class multijs (no line info...)
#36  exec(Context, Scriptable) in class multijs (no line info...)
#37  run(Context) in file org/mozilla/javascript/optimizer/OptRuntime.java:242
#38  call(ContextFactory, ContextAction) in file org/mozilla/javascript/Context.java:540
#39  call(ContextAction) in file org/mozilla/javascript/Context.java:457
#40  main(Script, String[]) in file org/mozilla/javascript/optimizer/OptRuntime.java:230
#41  main(String[]) in class multijs (no line info...)
```

FIG. 7.6: Contenu de la pile d'exécution avant filtrage

7.3.1 Blocs d'activation virtuels pour Rhino

```
(100
 (("doTopCall" any any "org.mozilla.javascript.ContextFactory")
  ("doTopCall" any any "org.mozilla.javascript.ScriptRuntime")
  ("call" ("Context" "Scriptable" "Scriptable" "Object[]") any (? any))
  ("exec" ("Context" "Scriptable") any (\ 1))
  ("run" any any "org.mozilla.javascript.optimizer.OptRuntime$1")
  (* ("call" any any "org.mozilla.javascript.optimizer.Context"))
  ("main" any any "org.mozilla.javascript.optimizer.OptRuntime")
  ("main" ("java.lang.String[]") any (\ 1)))
 mask)

(100
 ((["^_c[0-9]+$"] ((? any) . (? any)) "java.lang.Object" (\ 1))
  ("call" (\ 2) "java.lang.Object" (\ 1))
  (* (["^call.*"] any any "org.mozilla.javascript.optimizer.OptRuntime")))
 sf-rhino-function)

(100
 (("invoke0" any any (? sf-rhino-reflect))
  (* ("invoke" any any (\ 1)))
  ("invoke" any any "java.lang.reflect.Method")
  ("invoke" any any "org.mozilla.javascript.MemberBox")
  ("call" any any "org.mozilla.javascript.NativeJavaMethod")
  (["^call.*"] any any "org.mozilla.javascript.optimizer.OptRuntime"))
 mask)
```

FIG. 7.7: Ensemble de règles de remplacement pour Rhino

Dans le contenu de la pile présentée dans la figure 7.6, les fonctions Rhino sont situées entre les blocs d'activation 41 et 21. Les blocs d'activation provenant de la classe `multijs` illustrent le fait que le fichier ECMAScript utilisateur a directement été compilé en code-octet JVM. Les autres blocs d'activation montrent que la bibliothèque d'exécution de Rhino est écrite en Java. Le masquage de ces blocs est pris en charge par les règles de remplacement présentées dans la figure 7.7.

Un premier motif de code synthétique est formé par les blocs 41 à 33. Il représente le démarrage d'un programme Rhino. Le point d'entrée JVM `main` est en charge de l'initialisation de la bibliothèque d'exécution de Rhino (bloc 40). Ensuite, cette bibliothèque appelle la fonction principale du programme utilisateur (bloc 36), dont le but est d'évaluer les expressions globales du programme (blocs 34 et 33). La première règle de remplacement de la figure 7.7 est chargée de faire disparaître ces blocs d'activation dans la vue virtuelle.

Plus haut dans la pile, les blocs 32 à 31 et les blocs 30 à 28 représentent le motif responsable de l'implantation de l'appel de fonction d'ordre supérieur, tel qu'il a été décrit dans la section 7.1.1. La seconde règle de remplacement de la figure 7.7 permet de masquer les fonctions intermédiaires de ce motif et la fonction de remplacement `sf-rhino-function` est en charge de reconstruire les noms de fonctions utilisateur à partir des _c*n*.

Le dernier motif de code Rhino s'étend du bloc 27 au bloc 21 et représente les appels de fonctions étrangères à la plateforme d'exécution Rhino. Ce type d'appel n'est pas résolu au moment de la compilation, il est implanté à l'aide de l'API de réflexivité de Java. La troisième règle de remplacement de la figure 7.7 permet de masquer tout ces blocs

d'activation. Dans le bloc 21, la fonction de remplacement `sf-rhino-reflection` effectue une détection *ad-hoc* : elle identifie l'implanteur de l'API de réflexivité (par exemple Sun, IBM, GNU...) et met à jour la mémoire de recherche pour spécifier le nom du package JVM d'où doit provenir la prochaine fonction dans la pile. Cette « gymnastique » est nécessaire car la pile est explorée en partant du sommet.

7.3.2 Blocs d'activation virtuels pour Jython

```
(100
  ((["^(.*)\\$[0-9]+$"] any any (? any))
   ("call_function" any any (\ 2))
   (* ("call" any any ["^org.python.core.*"]))
   (* ("__call__" any any ["^org.python.core.*"]))
   ("_jcallexc" any any "org.python.core.PyObject")
   ("_jcall" any any "org.python.core.PyObject")
   (["\\1"] any any any))
  sf-jython-signature-function)

(100
 (("invoke0" any any (? sf-python-reflect))
  (* ("invoke" any any (\ 1)))
  ("invoke" any any "java.lang.reflect.Method")
  (* ("__call__" any any ["^org.python.core.*"]))
  ("invoke" ("java.lang.String") any "org.python.core.PyInstance"))
 mask)
```

FIG. 7.8: Ensemble de règles de remplacement pour Jython

Les blocs d'activation Jython sont situés dans la pile entre le bloc 20 et le bloc 2. Le contenu de la pile indique que toutes les fonctions représentées par ces blocs d'activation proviennent de fichiers Java, ce qui prouve que le code source Jython est d'abord traduit en un code Java avant d'être compilé. Cette séquence de blocs contient deux motifs de code synthétique.

Le premier motif de code Jython s'étend du bloc 20 au bloc 10. Le bloc 20 correspond à la fonction `gee` telle qu'elle est exportée dans le code source `multipy.py`. Dans les blocs d'activation 19 à 11, la plateforme d'exécution Jython enrobe les arguments étrangers pour qu'ils puissent être utilisés par du code Jython. Enfin, le bloc 10 représente la fonction produite contenant le code réel de la fonction `gee`. Il est à noter qu'une fois l'enrobage effectué, le motif effectue un appel de fonction classique (entre les blocs 15 et 10). La première règle de remplacement de la figure 7.8 transforme l'ensemble de ces blocs. Le premier filtre de bloc conserve en mémoire le nom de la fonction du bloc 10 jusqu'au caractère `$`. Le second filtre de bloc vérifie que le fonction du bloc suivant provient de la même classe JVM que la fonction du premier bloc du motif. Les quatre filtres de blocs suivants détectent les fonctions intermédiaires provenant de la bibliothèque Jython. Le dernier filtre de bloc vérifier que le nom de la fonction du dernier bloc du motif correspond au nom conservé en mémoire. La fonction `sf-jython-signature-function` remplace toutes ces fonctions par un bloc d'activation virtuel qui correspond à la fonction `gee` dans le code source.

Le dernier motif de code Jython correspond à l'implantation de l'appel des fonctions étrangères depuis la plateforme d'exécution Jython. Comme dans la plateforme Rhino, ce

type d'appel est implanté à l'aide de l'API de réflexion Java. On peut remarquer que durant la construction de la vue virtuelle, les règles de remplacement détectant ces deux types de motifs ne sont pas en conflit puisqu'elles terminent différemment.

En appliquant les différentes règles de remplacement présentées dans cette section, la construction de la vue virtuelle retourne une représentation correcte de la pile logique[2]. Si l'on ajoute à cela une exécution pas-à-pas adaptée aux bibliothèques d'exécution Rhino et Jython, cela permet au débogueur de retrouver toute son efficacité.

```
#0  (<anonymous:1003:multiscm.scm:4>) in file multiscm.scm:6
#1  gee(obj, obj) in file multipy.py (no line info...)
#2  foo() in file multijs.js:5
#3  <toplevel-init>() in file multijs.js:8
```

7.4 Expérience acquise avec les règles de remplacement

Les différentes expériences menées dans ces travaux ont montré que la manière d'extraire des informations de débogage en explorant la pile est très différente selon l'implantation du langage et son schéma de compilation. Par exemple, dans le cas de Bigloo, le compilateur transforme le code Scheme en des fonctions JVM de manière directe. Pour que la construction des blocs d'activation virtuels fonctionne même lorsqu'une classe n'est pas compilée en mode débogage, le compilateur doit inclure suffisamment d'« indices » dans le nom des fonctions produites. Cette approche rend l'ensemble de règles de remplacement très dépendant de propriétés syntaxiques, ce qui peut être gênant en cas de modification du compilateur. A contrario, dans la plateforme Jython, le compilateur produit des informations de débogage *ad-hoc* conservées dans des champs de classes JVM. Ces informations sont systématiquement produites et accessibles à l'aide de l'interprète du langage. Par conséquent, les règles de remplacement à concevoir sont moins complexes.

L'utilisation de langages complets comme Bigloo, Rhino et Jython a mis en évidence le fait que certaines règles de remplacement sont délicates à mettre en œuvre. Par exemple, pour masquer les motifs de code utilisant l'API de reflexivité Java, il faut coder « en dur » le nom de certains implanteurs de l'API (comme Sun ou IBM). De même, pour masquer les appels de fonctions Rhino, il ne faut pas utiliser la valeur des arguments pour ne pas dépendre des informations de débogage. Dans la pratique, du fait que la conception des règles de remplacement incombe aux personnes en charge du développement de ces compilateurs, la difficulté globale de mise en œuvre reste tout à fait raisonnable.

L'exemple de débogage multi-langage a montré que la construction de pile virtuelle reste effective en présence de plusieurs langages et donc de plusieurs ensembles de règles de remplacement. Par exemple, même si les plates-formes Jython et Rhino produisent du code similaire pour les appels de fonctions étrangères, le contexte de la détection de motif est suffisant pour éviter toute ambiguïté dans le choix du remplacement. Dans la pratique, ce risque d'ambiguïté est directement dépendant de la précision des règles de remplacement utilisées durant la session de débogage.

[2]Il n'y a pas d'information de ligne dans le bloc 1 car à ce jour, le compilateur Jython ne la produit pas pour les fonctions compilées.

7.5 Conclusion

Ce chapitre a présenté des exemples concrets d'utilisation des règles de remplacement sur deux langages de haut niveau : Rhino, un compilateur de programmes ECMAScript qui produit du code-octet JVM et Jython, une plateforme d'exécution de programmes Python pour la JVM. Le chapitre a aussi décrit en détail un exemple de débogage de programme multi-langages mêlant Bigloo, Rhino et Jython.

Les supports de débogage pour les plates-formes Rhino et Jython présentés dans ce chapitre ne sont pas encore complet. Il serait nécessaire de permettre la pose de points d'arrêt pour Jython, de prendre en charge les fonctions interprétés créés par l'évaluateur ou de fournir une exécution pas-à-pas correcte quel que soit le contexte d'exécution. Toutefois, les expériences menées dans ce chapitre ont validées les mécanismes de représentations virtuelles développés dans Bugloo. Tout d'abord, elles ont montrées que ces mécanismes étaient applicables à différents types de langage de haut niveau et qu'ils étaient efficaces quelle que soit le schéma de compilation employé. De plus, elles ont montré que les mécanismes développés pouvaient être utilisés sans problème pour déboguer des programmes multi-langages, ce qui était une exigence majeure énoncée au début de ce manuscrit.

Chapitre 8

Étendre Bugloo : le débogage des Fair Threads

Il existe deux grandes politiques d'ordonnancement de programmes concurrents : l'ordonnancement *préemptif* et le *coopératif*. Le premier est connu pour être difficile à déboguer car il est en général non déterministe, ce qui engendre des bogues complexes et très difficile à détecter de manière systématique. Le second est un modèle plus simple à débogueur.

Le langage Bigloo fournit une bibliothèque de programmation concurrente basée sur un ordonnancement déterministe et sur la programmation réactive synchrone. Ce chapitre présente une série d'outils développés pour permettre le débogage de cette bibliothèque. Il montre aussi comment utiliser l'API de Bugloo pour implanter ce genre d'extensions de débogage.

8.1 La programmation concurrente et le débogage

Les plates-formes d'exécution modernes permettent d'exécuter des programmes « multi-tâches » : au sein d'une application unique, il peut ainsi exister plusieurs flots d'exécution virtuels indépendant appelés *threads*. Ces derniers sont couramment utilisés dans les programmes de nos jours.

La programmation concurrente est une tâche difficile : tout d'abord, parce qu'entremêler des flots d'exécution est une tâche intrinsèquement complexe ; ensuite, parce que les types de bogues causés par les programmes multi-tâches sont en général difficiles à prendre en charge avec des débogueurs traditionnels.

Il existe différents modèles de programmation concurrente. Les différentes approches peuvent être regroupées en deux grandes catégories : celles utilisant un ordonnancement préemptif et celles utilisant un ordonnancement coopératif. Chacune de ces catégories offre des avantages et des inconvénients du point de vue du débogage.

8.1.1 Ordonnancement préemptif

L'ordonnancement préemptif est apparu dans les systèmes d'exploitations à la fin des années 70 avec le modèle des PThreads [NBF96]. Cette approche s'est ensuite généralisée dans les langages de programmation au milieu des années 90. Dans les PThreads, la bibliothèque de *thread* (pouvant être le système d'exploitation sous-jacent) peut suspendre

l'exécution d'un *thread* à tout moment pour en ordonnancer un autre. En général, ce genre de modèle tire parti des architectures multi-processeurs (SMP ou SMT). Malheureusement, les implantations de ce modèle induisent un ordonnancement non-déterministe, ce qui complique la conception des programmes et rend leur débogage « douloureux » :

- les bogues liés à la concurrence se produisent de manière aléatoire du fait du non-déterminisme de l'ordonnanceur. Le simple fait d'ajouter des `prints` dans le programme ou de le déboguer suffit souvent à faire disparaître un bogue. On appelle cela un *Heisenbug* [Bou04], en référence au fameux principe d'incertitude de Heisenberg[1] ;
- pour accéder à une ressource partagée, les *threads* acquièrent des *verrous* pour garantir une exclusion mutuelle entre eux. L'oubli d'une prise de verrou peut entraîner des corruptions mémoires, auquel cas le débogueur n'est plus d'aucune utilité ;
- l'ordonnancement des *threads* peut mener à un état de blocage collectif appelé *verrou mortel*. Cela peut se produire lorsque différents *threads* sont en concurrence pour l'acquisition d'un même verrou. La cause du bogue est difficile à comprendre sans connaître l'ordonnancement qui a mené au blocage ;
- les communications et les synchronisations entre *threads* s'effectuent par émission de *notifications*. Elles peuvent échouer si un *thread* n'est pas encore à l'écoute au moment de l'émission. Là encore, le débogueur n'est pas utile car il ne peut pas inspecter l'ordonnancement fautif ;
- certaines bibliothèques permettent de fixer les priorités d'ordonnancement des *threads*. Ces propriétés peuvent entraîner des problèmes d'inversion de priorités [SRL90] que les débogueurs n'aident pas à résoudre.

Certains modèles de programmation ne souffrent pas de tous les problèmes précédents. Par exemple, CML [Rep91] et Erlang [Arm97] restreignent la communication entre les *threads* à des envois de messages à travers des canaux. Ainsi, les corruptions mémoires sont évitées et les messages envoyés sont toujours reçus. En revanche, dans ces modèles, il n'y a plus réellement de mémoire partagée et la survenue de verrous mortels est toujours possible.

L'idée d'employer des traces pour comprendre la cause d'un verrou mortel ou d'une corruption mémoire n'est pas envisageable dans la pratique du fait de la nature trop aléatoire de la survenue de ce type de bogue. Les utilisateurs sont donc contraints d'utiliser des outils basés sur des analyses statiques [Car97, FF00b, WTE05] pour détecter des sources potentielles d'erreurs dans leur programme. Lorsque le programme le permet, ils peuvent utiliser des *model checkers* [HP98], des outils se servant de logiques temporelles et d'exploration d'espace d'état pour exhiber des séquences d'exécution menant à des verrous mortels.

8.1.2 Ordonnancement coopératif

L'ordonnancement coopératif est un modèle plus ancien, dans lequel c'est aux *threads* eux-même de coopérer, c'est-à-dire de rendre la main à l'ordonnanceur pour qu'un autre *thread* puisse s'exécuter. C'est un modèle déterministe dans lequel un seul *thread* s'exécute à la fois (ce qui empêche de profiter des multi-processeurs). Ce type d'ordonnancement élimine les deux sources de bogues les plus problématiques du modèle préemptif : les corruptions mémoires causés par des problèmes d'exclusions mutuelles et l'impossibilité de rejouer des exécutions à cause du non-déterminisme. De plus, il conserve la propriété de mémoire partagée.

[1] Ce principe décrit l'impossibilité de mesurer à la fois la position d'une particule et sa vitesse de façon exacte.

Dans les bibliothèques d'ordonnancement coopératif, les problèmes de synchronisation comme les verrous mortels peuvent toujours se produire. A cela viennent s'ajouter des types de bogues spécifiques à ce modèle :

- si un *thread* oublie de coopérer, les autres *threads* du programme ne peuvent jamais s'exécuter ;
- trop peu de coopérations peut entraîner des problèmes d'interactivité, par exemple dans le contexte des programmes à interfaces graphiques ;
- trop de coopérations peut entraîner des problèmes de performances. C'est un problème similaire à la prise excessive de verrous pour protéger des variables partagées dans le modèle préemptif.

Dans le modèle de programmation coopératif, un débogueur peut employer des techniques de traces afin de détecter la cause des bogues liés aux problèmes de coopérations. En effet, les exécutions étant déterministes, un programme défaillant peut être facilement être rejoué dans le débogueur afin de le tracer.

8.2 Le modèle de programmation des Fair Threads

La plateforme Bigloo propose un modèle alternatif de programmation concurrente, appelé *Fair Threads* [SBS04]. Ce modèle, fortement inspiré de la programmation réactive synchrone, a les caractéristiques suivantes :

- il fournit un ordonnanceur *coopératif* et *déterministe*. Ce dernier divise l'exécution en *instants*, des tours d'exécution logiques durant lesquels chaque *thread* a été exécuté. Dans ce modèle, il n'est pas nécessaire d'acquérir des verrous pour protéger les accès aux données. De plus, une exécution peut être rejouée à l'identique à volonté ;
- Les *threads* communiquent entre eux en diffusant des *signaux* dans l'ordonnanceur. La sémantique garantie que la présence d'un signal diffusé au cours d'un instant pourra être détectée par tous les *threads* jusqu'à la fin de cet instant ;
- Il offre la possibilité d'effectuer des entrées / sorties de manière asynchrones, sans bloquer l'ordonnancement des autres *threads* et en profitant des architectures multiprocesseurs. Le non-déterminisme résultant de ces opérations asynchrones est confiné dans des parties clairement définies du programme.

Le modèle de communication proposé dans les *Fair Threads* ne fait pas disparaître tous les types de bogues liés à la programmation concurrente. Cependant, les bogues inhérents à ce modèle sont différents de ceux rencontrés dans le modèle classique de programmation concurrente à mémoire partagée que propose les PThreads. Tout d'abord, il n'y a plus de problème de corruption de données car les *Fair Threads* ne sont pas préemptés par l'ordonnanceur de manière non-déterministe. De plus, les bogues inhérents à ce modèle découlent tous de problèmes de coopération ou de communication au cours d'un instant :

- si un *thread* oublie de coopérer, tout l'ordonnancement est bloqué. C'est un bogue classique des modèles de programmation coopératifs ;
- contrairement aux PThreads, une synchronisation n'échoue pas si une diffusion est émise avant qu'un thread se soit mis en attente. En revanche, elle peut échouer si la diffusion et la détection n'ont pas lieu dans le même instant ;
- même sans exclusion mutuelle, l'exécution du programme peut entraîner un verrou mortel. Cela se produit typiquement lorsqu'une synchronisation nécessitant la diffusion de plusieurs signaux s'étale de manière imprévue sur plusieurs instants.

En résumé, les *Fair Threads* introduisent de nouveaux types de bogues spécifiques aux primitives de communication mises en jeu, mais permettent un débogage beaucoup plus simple : au lieu de s'assurer de la correction des exclusions mutuelles ou des synchronisations effectuées dans le programme, le débogage consiste à inspecter l'état d'un ordonnanceur aux cours d'instants afin de retrouver la cause d'un bogue.

Bugloo étant un débogueur extensible, un module spécifique aux *Fair Threads* a été développé pour fournir des outils de débogage qui tiennent compte des spécificités de ce modèle de programmation :

- il ajoute de nouvelles commandes permettant d'étendre la granularité de l'exécution pas-à-pas, comme par exemple l'exécution jusqu'au prochain point de coopération ou à la prochaine diffusion de signal ;
- il propose des nouvelles vues pour l'inspecteur structurel (cf chapitre 3.1.2), afin de visualiser efficacement l'état de l'ordonnanceur, des *threads* ou des signaux émis au cours de l'instant. Ce type d'information permet de détecter des bogues simples comme les verrous mortels ou les défauts de coopération ;
- il permet de tracer l'ordonnancement des *threads*, ainsi que la diffusion et la réception des signaux tout au long de l'exécution. La trace produite peut ensuite être analysée de manière graphique. C'est un moyen efficace de débogueur des protocoles de communication complexes s'étalant sur plusieurs instants.

La suite de ce chapitre est organisée comme suit : tout d'abord, le modèle de programmation concurrente des *Fair Threads* est présente en détail, ainsi que sa mise en œuvre dans Bigloo. La section suivante présente les fonctionnalités de débogage principales, à savoir l'exécution pas-à-pas étendue et l'inspecteur de *threads*. Une autre section décrit l'outil d'enregistrement et de visualisation de l'ordonnancement des *threads*. Puis, un exemple concret vient illustrer l'utilisation de tous ces outils. Enfin, la fin du chapitre décrit brièvement l'utilisation de l'API de Bugloo pour implanter cette extension de débogage, ainsi que l'expérience acquise durant l'utilisation de ces nouveau outils.

8.3 Le modèle de programmation des Fair Threads

Cette section détaille les caractéristiques du modèle de programmation concurrente des *Fair Threads*. La description de l'ordonnancement d'un programme abstrait vient illustrer le fonctionnement de ce modèle. La suite de la section détaille l'API des *Fair Threads* accessible dans le langage Bigloo. La fin de la section présente les différents types de bogues engendrés par ce modèle de programmation.

8.3.1 Principe d'ordonnancement des Fair Threads

Dans le modèle des *Fair Threads*, chaque *thread* est une entité particulière disposant de sa propre pile d'exécution et de son environnement dynamique[2]. Les *threads* sont *attachés* à un ordonnanceur coopératif, dans lequel un seul d'entre eux peut s'exécuter à la fois. Lorsqu'un *thread* coopère, l'ordonnanceur donne le contrôle à un autre *thread*. L'ordonnanceur est lui-même un *fair thread*. La sémantique des *Fair Threads* définit clairement le comportement de l'ordonnanceur, ainsi que celui de la diffusion et de la réception des signaux :

[2] Cet environnement contient par exemple la sortie standard du *thread* ou l'état courant du lecteur Scheme.

– l'ordonnancement est décomposé en unités temporelles logiques appelées *instants*. Au cours d'un instant, les *threads* peuvent communiquer en attendant ou en diffusant des *signaux* dans l'ordonnanceur. Un signal reste présent jusqu'à la fin de l'instant et peut être valué ;
– si un signal est diffusé au cours d'un instant, tous les *threads* en attente de ce signal ont la garantie d'être réveillés avant la fin de l'instant. En particulier, si un *thread* se met en attente d'un signal déjà diffusé au cours de l'instant, il est immédiatement notifié et continue son exécution ;
– un signal peut être diffusé plusieurs fois au cours de l'instant. Les *threads* ont la possibilité de se faire réveiller par l'ordonnanceur à l'instant suivant avec la liste des valeurs émises avec le signal durant l'instant écoulé ;
– un *thread* peut être réordonnancé dans l'instant courant si un signal a été diffusé depuis sa dernière élection. Un instant termine lorsque tous les *threads* ont été exécutés et qu'aucun nouveau signal n'a été diffusé.

8.3.2 Utilisation des Fair Threads

thread A

```
1 await sig1
2 await sig2
3 yield
4 await sig1
```

thread B

```
1 broadcast sig1
2 yield
3 broadcast sig3
```

thread C

```
1 await sig1
2 broadcast sig2
3 await sig3
```

FIG. 8.1: Exemple de communication entre *Fair Threads*

La figure 8.1 présente un exemple de programme abstrait qui illustre l'ordonnancement des *Fair Threads*. La progression de l'exécution est décomposée ci-dessous :

Instant 1 : le *thread* A s'endort à la ligne 1 en attendant que le signal `sig1` soit diffusé. L'ordonnanceur passe la main au *thread* B. Ce dernier diffuse le signal `sig1` à la ligne 1 et coopère explicitement à la ligne 2, ce qui termine son exécution pour l'instant. L'ordonnanceur élit ensuite le *thread* C. Ce dernier ne bloque pas sur l'attente du signal `sig1` car il a déjà été émis dans l'instant. Puis, il diffuse le signal `sig2` et s'endort en attente du signal `sig3`. A ce moment, l'instant n'est pas fini : l'ordonnanceur doit redonner la main au *thread* A car le signal `sig1` est présent. Ce thread, réveillé, continue son exécution sans bloquer sur l'attente de `sig2`, puis coopère explicitement à la ligne 3. À ce moment précis, tous les *threads* sont bloqués et aucun nouveau signal n'a été diffusé. Cela marque donc la fin de l'instant. *Tous les signaux sont réinitialisés.*

Instant 2 : le *thread* A se bloque à la ligne 4 en attente du signal `sig1` car il n'a pas encore été diffusé durant l'instant. Le *thread* B diffuse le signal `sig3` puis termine son exécution. Le *thread* C est réveillé par la présence de ce signal à la ligne 3 puis termine son exécution. À ce moment précis, le *thread* A est toujours bloqué, l'instant se termine et les signaux sont réinitialisés.

Instant 3 : le *thread* A est bloqué indéfiniment en attente de la diffusion du signal `sig1`.

```
1  (define (funA)
2     (thread-await! 'sig1)
3     (thread-await! 'sig2)
4     (thread-yield!)
5     (thread-await! 'sig1))
6
7  (define (funB)
8     (broadcast! 'sig1)
9     (thread-yield!)
10    (broadcast! 'sig3))
11
12 (define (funC)
13    (thread-await! 'sig1)
14    (broadcast! 'sig2)
15    (thread-await! 'sig3))
16
17 (define (main args)
18    (thread-start! (make-thread funA "fairthread A"))
19    (thread-start! (make-thread funB "fairthread B"))
20    (thread-start! (make-thread funC "fairthread C"))
21    (scheduler-start!))
```

FIG. 8.2: Implantation de l'exemple de la figure 8.1

8.3.3 L'API des Fair Threads

L'API des *Fair Threads* a été conçu pour pouvoir être utilisée en lieu et place de la bibliothèque de *threads* SRFI-18 [Fee00]. Cette bibliothèque est une extension Scheme standard pour la programmation concurrente inspirée des PThreads. L'exemple de la figure 8.2 présente un programme Bigloo équivalent au programme abstrait de la figure 8.1. La suite de la section décrit les fonctions principales de la bibliothèque des *Fair Threads*.

Manipulation de threads

Comme le montre la figure 8.2 à la ligne 18, un *thread* est crée à l'aide de la fonction `(make-thread` *`thunk`* `.` *`name`*`)`, où *`thunk`* représente la fonction à exécuter et *`name`* le nom optionnel donné au *thread*. Le *thread* doit être démarré à l'aide de la fonction `thread-start!` pour être pris en compte par l'ordonnanceur.

La coopération est effectuée par un appel à la fonction `thread-yield!` (lignes 4 et 9 de la figure). Un *thread* peut forcer la terminaison d'un autre *thread* à la fin de l'instant à l'aide de la fonction `(thread-terminate!` *`thread`*`)`.

Dans les *Fair Threads*, l'ordonnanceur doit être démarré explicitement par un appel à la fonction `scheduler-start!`. Une fois démarré, ce dernier s'exécute jusqu'à ce que tous ses `thds` soient terminés ou ne puissent plus progresser.

Communication entre threads

Un *thread* diffuse des signaux dans l'ordonnanceur à l'aide de la fonction `(broadcast` *`sig`* `.` *`value`*`)`. Le signal *`sig`* est un objet Scheme arbitraire qui peut être associé à une valeur optionnelle qui sera reçue par les *threads* en attente lors de leur réveil.

Un *thread* peut attendre la présence d'un signal au moyen de la fonction `(thread-await!` *`sig`*`)` ou bien attendre plusieurs signaux à la fois à l'aide de la fonction `(thread-await!*`

`sig`). Enfin, un *thread* a la possibilité de récupérer l'ensemble des valeurs émises pour chacune des émissions d'un signal au moyen de la fonction `(thread-get-values sig)`. Dans ce cas, le *thread* s'endort jusqu'à la fin de l'instant et l'ordonnanceur le réveille à l'instant suivant en lui retournant la liste des valeurs.

Par souci de compatibilité, l'API des *Fair Threads* fournit les primitives SRFI-18 comme les mutex ou les variables de conditions. En interne, ces primitives sont implantées à l'aide de signaux. Elles ne sont pas utiles dans un ordonnanceur coopératif et ne sont donc pas traitées dans ce chapitre.

8.3.4 Bogues rencontrés dans le modèle des Fair Threads

Avec les *Fair Threads*, les communications ou les synchronisations reposent sur des signaux et des instants au lieu des traditionnels mutex et variables de conditions. Les types de bogues pouvant se produire dans ce modèle peuvent être regroupés en deux grandes catégories :

- les bogues simple comme les problèmes de coopération dans *l'instant courant*. Il s'agit typiquement de situations dans lesquelles un *thread* est chargé d'exécuter un calcul en réponse à la diffusion d'un signal. Lorsque le *thread* oublie de coopérer, la boucle se répète indéfiniment car le signal est toujours présent dans l'instant. Pour traquer ce genre d'erreur, l'extension de débogage fournit à l'utilisateur une exécution pas-à-pas étendue et un inspecteur d'ordonnanceur. Ces deux outils sont présentés dans la section 8.4 ;
- les bogues ne pouvant être résolus qu'en se rappelant de l'état de l'ordonnanceur *plusieurs instants auparavant*. C'est le cas par exemple des synchronisations entraînant des verrous mortels. Ce genre de problème ne peut pas survenir avec des signaux s'ils sont diffusés dans l'instant. En revanche, il peut survenir si un *thread* attend un signal qui a été diffusé dans un instant précédent. Pour régler ce genre d'erreur, l'extension de débogage fournit un outil permettant de tracer l'ordonnanceur. Il est présenté dans la section 8.5.

8.4 Débogage des Fair Threads

Cette section présente les deux premiers outils de débogage des *Fair Threads* : l'exécution pas-à-pas étendue et l'inspecteur d'ordonnanceur. Ces outils peuvent être utilisés conjointement pour comprendre la cause des bogues qui peuvent se produire au cours de l'instant, comme les problèmes de communications ou les défauts de coopérations.

8.4.1 La boîte à outils de débogage des Fair Threads

Les fonctionnalités fournies par l'extension de débogage sont toutes accessibles à partir d'une boîte à outils graphique. Supposons que le programme d'exemple de la figure 8.2 soit en cours de débogage et que l'exécution soit suspendue à la ligne 13. L'utilisateur peut alors faire apparaître la boîte à outils depuis la ligne de commande :

```
(bugloo) (gui show fairthreads)
```

La boîte à outils, présentée dans la figure 8.3, permet d'accéder aux outils de débogage spécifiques aux *Fair Threads*. De haut en bas, la boîte contient une série de boutons qui étendent l'exécution pas-à-pas, des boutons pour manipuler les traces de l'ordonnanceur (cf.

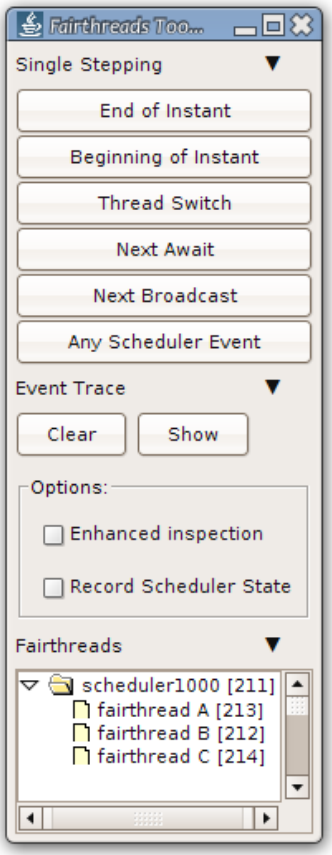


FIG. 8.3: La boîte à outils des *Fair Threads*

section 8.6) et la liste des *fair threads* présents dans le programme. La suite de la section décrit ces outils.

Exécution pas-à-pas étendue

L'utilisation de signaux et d'instants dans le code introduit des nouveaux points temporels logiques dans l'exécution des programmes. L'extension de débogage des *Fair Threads* les prends en compte en ajoutant six nouvelles commandes d'exécution pas-à-pas accessibles par les boutons de la boîte à outils :

End of Instant continue l'exécution jusqu'à la fin de l'instant courant et suspend l'exécution juste avant le début de l'instant suivant. Cela permet d'inspecter l'état de tous les *threads* en fin d'instant et la liste des signaux émis ;

Beginning of Instant suspend l'exécution dès qu'un nouvel instant est commencé. Ce saut permet d'avancer rapidement jusqu'au début d'un instant utile pour le débogage et de procéder ensuite à une exécution pas-à-pas plus fine ;

Thread Switch suspend l'exécution dès qu'un nouveau *thread* prend le contrôle de l'exécution. Cela permet d'examiner l'ordonnancement des *threads* durant l'instant ;

Next Await continue l'exécution jusqu'à ce qu'un *thread* attende un signal. Ce saut permet d'inspecter les communications effectuées dans le programme ;

Next Broadcast continue l'exécution jusqu'à ce qu'un *thread* diffuse un signal. Ce

saut est le dual du saut précédent ;

Any Scheduler Event suspend l'exécution lors de la survenue de l'un des événements précédents.

Tracer les événements survenant dans l'ordonnanceur

L'extension de débogage pour les *Fair Threads* permet d'enregistrer les événements se produisant dans l'ordonnanceur comme les changements de contexte, les diffusion de signaux ou les fins d'instants. L'utilité d'enregistrer ces événements est double. Tout d'abord cela permet d'améliorer les informations de débogage fournies à l'utilisateur. Par exemple, lors de l'inspection de l'ordonnanceur, le débogueur peut afficher le *thread* responsable de la diffusion d'un signal, ainsi que sa position dans le code source. De plus, la trace permet de « surveiller » l'exécution d'une partie du programme afin d'obtenir une vision détaillée de ce qui s'est produit dans l'ordonnanceur durant plusieurs instants.

Dans la boîte à outils, deux options permettent de contrôler l'enregistrement des événements :

Enhanced Inspection cette option active l'enregistrement des événements se produisant dans l'ordonnanceur. Quand l'utilisateur effectue un exécution pas-à-pas, il obtient automatiquement des informations de débogage étendues[3]. Cette inspection étendue est automatiquement débrayée lorsque l'exécution est résumée. De plus, les événements enregistrés sont effacés à chaque changement d'instant ;

Record Scheduler State avec cette option, l'enregistrement des événements reste activée tout au long de l'exécution et entre les instants. Plus tard, la trace produite peut être visualisée ou effacée avec les boutons correspondants dans la boîte à outils (cf. figure 8.3).

Liste des threads

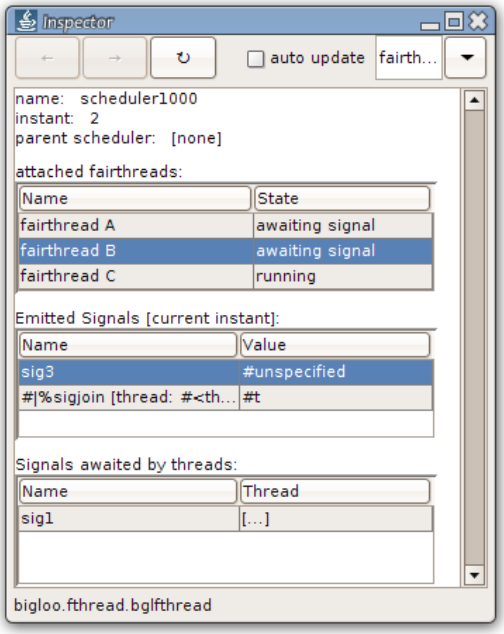
La dernière partie de la boîte à outil présente la liste des *fair threads* vivants dans le programme. Elle masque les *threads* JVM standard. Les *threads* sont présentés sous forme d'arbre dont les nœuds représentent des ordonnanceurs et les feuilles des *threads*. Il peut y avoir plusieurs ordonnanceurs dans le programme et ces derniers peuvent être emboîtés puisque ce ne sont eux aussi des *threads*.

La figure 8.3 montre les *threads* du programme d'exemple. Ils sont identifiés par leur nom et un numéro unique pour les référencer depuis la ligne de commande de Bugloo. Un double-clic sur un *thread* permet de lancer un inspecteur graphique. Les différents inspecteurs sont décrit dans la section suivante.

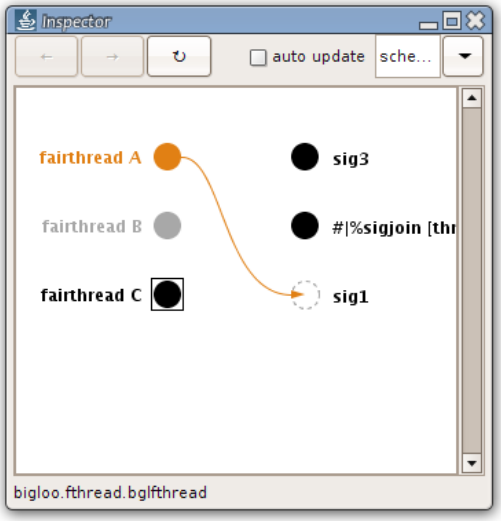
8.5 Inspecteurs spécialisés pour les Fair Threads

Cette section présente les nouvelles vues développées pour l'inspecteur structurel afin d'améliorer le débogage des *Fair Threads*. Ces vues servent à inspecter l'état courant de l'ordonnanceur et des *threads* qu'il contrôle. La suite de la sections décrit les deux vues développées pour l'ordonnanceur, la vue des *threads* et celle des signaux.

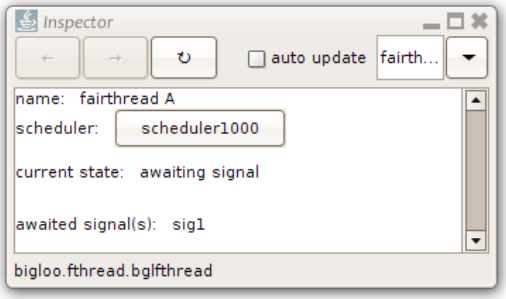
[3] Si l'option est activée au milieu d'un instant, elle ne sera pleinement effective qu'au début de l'instant suivant.



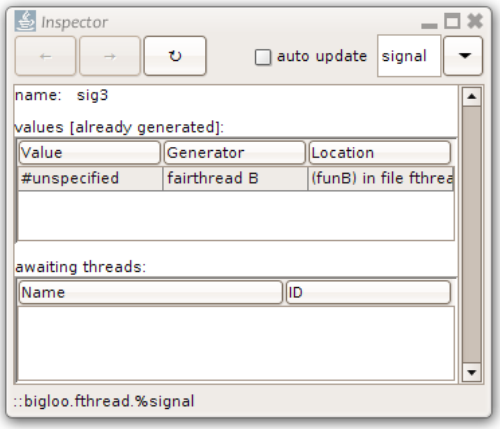
(a) L'inspecteur d'ordonnanceur



(b) L'inspecteur d'ordonnanceur en deux dimensions



(c) L'inspecteur de *thread*



(d) L'inspecteur de signal

FIG. 8.4: Nouvelles vues pour l'inspecteur structurel

8.5.1 Vues pour l'ordonnanceur

Présentation détaillée

La capture d'écran de la figure 8.4(a) détaille l'état de l'ordonnanceur lorsque le programme de la figure 8.2 est suspendu à la ligne 15. La vue est décomposée en quatre parties.

La première partie présente des informations sur l'ordonnanceur : son nom, l'instant courant et, le cas échéant, l'ordonnanceur auquel il est rattaché.

La seconde partie est une table listant les *threads* attachés à cet ordonnanceur. Les informations concernant un *thread* incluent son nom et son état courant. Contrairement

aux *threads* POSIX, un *fair thread* peut prendre six différents états :

running le *thread* est le celui en cours d'exécution dans l'ordonnanceur ;

standby le *thread* peut être exécuté dans l'instant ;

await le *thread* est en attente de signal ;

end of instant le *thread* a terminé son exécution pour l'instant courant ;

terminated le *thread* a terminé toute son exéction ;

unattached le *thread* n'a pas encore démarré car il n'a pas été attaché à un ordonnanceur. Cet état n'est visible que dans la vue d'un thread (cf. section 8.5.2.)

Un double-clic sur une ligne commande à l'inspecteur d'empiler une nouvelle vue d'inspection pour le *thread* sélectionné.

Enfin, les deux dernières parties de cette vue sont dédiées aux signaux :

- une première table contient la liste des signaux diffusés dans l'ordonnanceur durant l'instant. Chaque ligne contient le nom d'un signal et sa valeur associée (ou le symbole `[...]` si le signal a été diffusé plusieurs fois). Un double-clic empile une nouvelle vue pour le signal selectionné ;
- une deuxième table contient la liste des signaux attendus par les *threads* et qui n'ont pas encore été diffusés dans l'instant. Dès qu'un de ces signaux est diffusé, la ligne correspondante se déplace vers la table précédente.

Présentation graphique en deux dimensions

Le contenu d'un ordonnanceur peut aussi être représenté de manière purement graphique, en deux dimensions. La figure 8.4(b) donne un aperçu du rendu effectué dans cette vue. L'affichage s'inspire du débogueur Haskell CHD [CH04] et donne une vision plus synthétique que la vue précédente. Cette vue est décomposée en deux colonnes verticales : la colonne de gauche représente les *threads* contrôlés par l'ordonnanceur et la colonne de droite les signaux émis durant l'instant. La forme et la couleur des objets varient selon leur état. Un disque gris dans la colonne de gauche indique qu'un *thread* a terminé son exécution pour l'instant. Un disque noir encadré indique que le *thread* est en cours d'exécution. De même, un cercle en pointillé dans la colonne de droite indique qu'un signal est attendu par un ou plusieurs *threads*. Un cercle noir indique qu'il a déjà été diffusé dans l'instant. Enfin, les flèches pointées vers la colonne de droite indiquent quels sont les *threads* en attente de signaux. Des flèches pointées vers la gauche indiquent le réveil de *threads*.

8.5.2 Vue pour un thread

La vue spécialisée pour les *threads*, présentée en figure 8.4(c), produit un affichage simple. Elle présente tout d'abord le nom du *thread* et celui de l'ordonnanceur auquel il est attaché. Un clic sur ce dernier empile une nouvelle vue dans l'inspecteur.

La vue permet de connaître l'état courant du *thread*, parmi les six état décrit précédemment. Enfin, si le *thread* est en attente d'un ou plusieurs signaux, les noms sont inclus dans la liste en bas de la vue.

Dans la capture d'écran de la figure 8.4(c), on voit que le *thread* A du programme d'exemple attend le signal `sig1`. En utilisant plusieurs fenêtres d'inspection, il est possible de visualiser l'état de plusieurs *threads* en même temps.

8.5.3 Vue pour un signal

La vue pour un signal (figure 8.4(d) est composée du nom du signal en cours d'inspection et de deux autres tables. La première table contient la liste des différentes valeurs associées à chaque émission du signal durant l'instant. Si l'option d'inspection étendue est activée, la vue précise, pour chaque signal, le *thread* et l'endroit dans le code source d'où est partit la diffusion. Enfin, la seconde table liste le nom et l'identificateur de chaque *thread* en attente du signal en cours d'inspection. Si le signal a déjà été émis au cours de l'instant, cette table est vide.

8.6 Tracer les événements de l'ordonnanceur

La section précédente a décrit les outils développés dans cette extension de débogage pour aider à corriger les problèmes de communication ou de synchronisation qui peuvent se produire durant un seul instant. Ces outils ne sont pas adaptés pour comprendre les causes des bogues dont l'origine se situe manifestement un ou plusieurs instant « en amont » dans le temps.

Cette section présente un outil de trace qui offre un moyen efficace de visualiser l'état de l'ordonnanceur *durant* l'instant et *entre* les instants. Il donne à l'utilisateur une vision précise de l'ordre dans lequel les *threads* sont exécutés et des communications qu'ils effectuent.

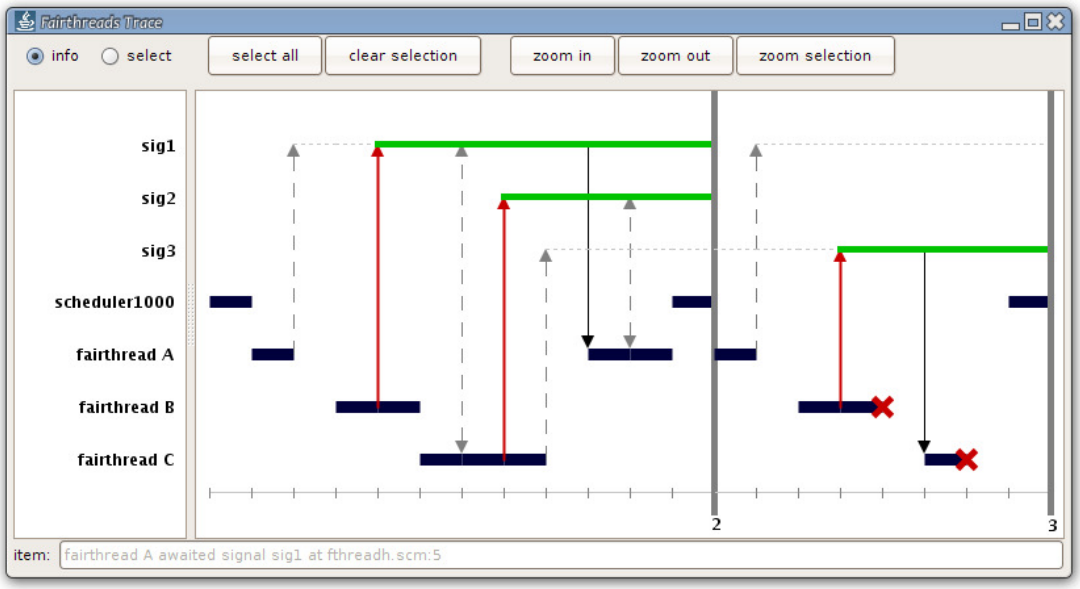


FIG. 8.5: Trace de l'ordonnancement du programme de la figure 8.2

Lorsque l'option de trace est actionnée dans la boîte à outils (cf. figure 8.3), l'utilisateur peut faire apparaître une fenêtre d'inspection graphique de la trace. À titre d'exemple, détaillons son utilisation sur le programme d'exemple de la figure 8.2. Lorsque l'exécution du programme atteint le point de verrou mortel, l'utilisateur peut suspendre l'exécution en tapant `CTRL+C` et demander à BUGLOO d'afficher la trace enregistrée. Le résultat est présenté dans la figure 8.5. La trace est affichée sous forme de graphe :

– l'axe vertical contient les *threads* attachés à l'ordonnanceur et tous les signaux diffusés durant la trace d'exécution enregistrée. Les signaux apparaissent toujours au sommet, suivi de l'ordonnanceur et des *threads* ;
– l'axe horizontal représente la progression de l'exécution durant les instants. Ces instants sont délimités par des lignes verticales épaisses et leur numéro est reporté en bas de ligne.

La trace est décomposée en unités logiques représentant des opérations atomiques qui se produisent dans l'ordonnanceur. Par exemple, une unité peut dénoter un changement de contexte, la diffusion d'un signal, l'attente d'un signal, la terminaison d'un *thread* ou bien encore l'exécution d'une entrée/sortie asynchrone. Le détail de la trace est expliqué ce-dessous.

Au début de l'instant 1, l'ordonnanceur `scheduler1001` a le contrôle de l'exécution. Cela est symbolisé par une marque horizontale épaisse. Puis, l'ordonnanceur donne la main au *thread* A, ce qui ajoute une ligne horizontale dans la trace.

Le *thread* A s'endort en attente du signal `sig1`. Cette attente est symbolisée par une flèche verticale pointillée pointant vers la ligne de vie du signal `sig1`. Une ligne horizontale pointillée est dessinée pour indiquer que ce signal n'a pas encore été diffusé dans l'instant.

Lorsque le contrôle passe au *thread* B, ce dernier diffuse `sig1`. La diffusion est symbolisée dans la trace par une flèche verticale épaisse pointant vers la ligne de vie de `sig1`. À partir de ce point dans la trace, la ligne de vie devient épaisse pour symboliser la présence du signal dans l'instant. Puis le *thread* coopère explicitement. Ces deux opérations logiques sont exécutées en séquence, ce qui explique la taille du segment horizontal associé au *thread* B.

Le *thread* C commence son exécution en attendant `sig1`. Comme ce signal est déjà présent, l'attente ne suspend pas son exécution. Ce phénomène est symbolisé dans la trace par une flèche pointillée à deux têtes.

Plus tard dans la trace, le *thread* A est réveillé car `sig1` a été émis. Ce réveil est représenté par une flèche vertical allant de la ligne de vie du signal jusqu'au *thread* A.

Au début de l'instant 2, tous les signaux présents dans l'ordonnanceur sont réinitialisés, les lignes de vie sont donc effacées. Puis, le *thread* A s'endort en attente du signal `sig1`.

En fin de trace, les deux croix diagonales à la fin des exécutions des *threads* B et C indiquent que ces *threads* ont terminé leur exécution et qu'ils ne seront donc plus jamais ordonnancés.

Il n'y a pas d'instant 3. En effet, le seul *thread* restant à ordonnancer est bloqué en attente d'un signal qui ne sera jamais diffusé. La fin de l'instant ne peut pas être atteinte, ce qui provoque le verrou mortel.

8.7 Utilisation des outils de débogage

Cette section présente un exemple pratique d'utilisation des outils de débogage de *Fair Threads* présentés précédemment. Elle décrit la manière dont Bugloo a pu être débogué durant son développement. Dans la suite, la section introduit l'implantation de la gestion des événements dans Bugloo, le bogue qu'elle contenait et la manière de le retrouver.

```
(define (fair-repl)
   (let loop ()
      (cond
         ((eq? status 'debuggee)
          (set! status #unspecified)
          (worker-start-action
           process-debuggee-events))
         ((eq? status 'process-input)
          (await-processable-event) )
         ((eq? status 'end-input)
          (set! status #unspecified) )
         ((eq? status 'stdin)
          (worker-start-action
           process-user-input))
         (else
          (check-print-prompt)
          (await-processable-event)))
      (loop)))
```

(a) *thread* principal du débogueur

```
(define (await-processable-event)
   (multiple-value-bind (_ sig)
      (thread-await*!
       '(debuggee stdin end-input))
      (set! status sig))
   (thread-yield!))
```

(b) mise en attente du *thread* principal

```
(define (worker-start-action action)
   (set! status 'process-input)
   (if (>= *workers-depth* *workers-length*)
       (push-worker))
   (broadcast!
    (list-ref *workers-stack* *workers-depth*)
    action))
```

(c) Exécution d'une action dans un nouveau *travailleur*

```
(define (worker-start-action-external action)
   (set! status 'process-input)
   (if (>= *workers-depth* *workers-length*) (push-worker))
   (scheduler-broadcast! (default-scheduler) (list-ref *workers-stack* *workers-depth*) action)
   (react-scheduler-until-done (default-scheduler)))
```

(d) Exécution d'une action dans un nouveau *travailleur*. Variante utilisée depuis un *thread* JVM standard (par exemple, celui de l'interface graphique)

```
(define (push-worker)
   (let* ((id (gensym 'worker)))
      (thread-start!
       (make-thread (lambda ()
                      (let loop ()
                         (let ((action (thread-await! id)))
                            (set! *workers-depth* (+ *workers-depth* 1))
                            (action)
                            (set! *workers-depth* (- *workers-depth* 1)))
                         (broadcast! 'end-input)
                         (thread-yield!)
                         (loop))) ))
      (string-append "Fair Worker " (integer->string *workers-length*))
      (set! *workers-stack* (append! *workers-stack* (list id)))
      (set! *workers-length* (+ *workers-length* 1))))
```

(e) Création d'un nouveau *travailleur*

FIG. 8.6: L'implantation *Fair Threads* de la gestion des événements dans le débogueur (cf. 3.6)

8.7.1 L'architecture de gestion d'événement dans Bugloo

Dans BUGLOO, la gestion de la JVM déboguée et des entrées utilisateur a progressivement évolué d'une simple fonction *mono-thread* à une architecture complète se servant de la bibliothèque des *Fair Threads*. Comme indiqué dans le chapitre 3, un *thread* principal écoute les différentes sources d'événements et aiguille leurs traitements vers des *threads* secondaires appelés *travailleurs*.

La figure 8.6 présente le cœur de l'implantation du mécanisme de gestion d'événements du débogueur. La fonction `fair-repl` (figure 8.6(a)) s'exécute dans le *thread* principal.

Par défaut, elle appelle la fonction `wait-processable-event` (figure 8.6(b)) qui attend à la ligne 3 qu'un événement extérieur soit diffusé dans l'ordonnanceur : entrée clavier, interaction graphique ou communication avec le débogué. Lorsque le *thread* est réveillé, il délègue le traitement de l'événement à un autre *thread* appelé un *travailleur*. Dans ce dernier, les appels distants dans la JVM du débogué sont effectués de manière asynchrone pour éviter de bloquer l'ordonnanceur coopératif (cf. 3.6, page 36).

La fonction `push-worker` (figure 8.6(e)) est en charge de créer des *travailleurs* et de les conserver dans une liste. Pour utiliser un *travailleur*, la fonction `worker-start-action` (figure 8.6(c)) diffuse un signal avec comme valeur associée l'action à exécuter. Pour utiliser un *travailleur* depuis un *thread* JVM standard (par exemple, celui de l'interface graphique), il faut appeler la fonction `worker-start-action-external`. Cette dernière est alors en charge de faire réagir l'ordonnanceur (ligne 5) après avoir diffusé le signal.

8.7.2 Débogage du débogueur

Dans les premières implantations de l'écoute des événements, un bogue bizarre se produisait : l'utilisateur ne pouvait plus se servir de certaines fonctionnalités graphiques comme l'inspecteur structurel, mais pouvait toujours en utiliser d'autres comme l'inspecteur d'allocations. Pour localiser la source du dysfonctionnement, il était nécessaire de déboguer le débogueur. Un point d'arrêt fut posé dans l'inspecteur structurel dans la fonction responsable de la construction des vues. Ce point d'arrêt n'était jamais atteint, ce qui semblait indiquer un problème avec les *travailleurs* ou d'autres *threads*.

Le problème de *threads* a du être pisté à l'aide des outils de débogage des *Fair Threads*. La démarche a tout d'abord consisté à demander l'utilisation de l'inspecteur structurel pour déclencher le bogue, puis à utiliser l'inspecteur de *threads* pour visualiser l'état dans lequel se trouvait l'ordonnanceur. Le résultat est affiché dans la figure 8.7. On y voit le *thread* principal nommé *Fair REPL* en attente d'événement, ainsi que deux *travailleurs* eux aussi en attente.

Le premier outil ne permettant pas de tirer des conclusions sur l'origine du bogue, une deuxième tentative a consisté à tracer l'ordonnanceur lorsque l'utilisateur demandait l'utilisation de l'inspecteur structurel. La trace obtenue est affichée dans la figure 8.8. On voit que l'action à la ligne de commande a provoqué l'émission du signal `stdin`[4], ce qui a réveillé le *thread* principal. Ce dernier coopère (cf figure 8.6(b), ligne 5) puis, à l'instant suivant, émet un signal pour réveiller le *travailleur* 0. Ce *travailleur* envoie à son tour un signal au *travailleur* suivant. Toutefois, à cet instant, le deuxième travailleur n'a pas encore été ajouté à l'ordonnanceur. La notification est donc perdue.

En analysant les informations de lignes produites par la trace, on peut voir que le *travailleur* 0 a émis son signal dans la fonction `worker-start-action`. De plus, l'exécution de cette fonction a eu pour effet de créer un nouveau *travailleur* (ligne 3). Or, la sémantique des *Fair Threads* indique qu'un *thread* nouvellement créé est ajouté à l'ordonnanceur seulement à l'instant suivant. L'outil de trace a donc permis de comprendre que le bogue provenait d'un oubli de coopération avant la ligne 5 de la fonction `worker-start-action`. On peut noter que `worker-start-action-external` ne nécessite pas de modification, car la fonction `scheduler-broadcast!` émet un signal dans l'ordonnanceur au début de l'instant suivant. Cela explique le fait que de nombreuses outils de débogage fonctionnaient correctement.

[4]Le disque vert indique que le signal a été émis par un *thread* JVM standard.

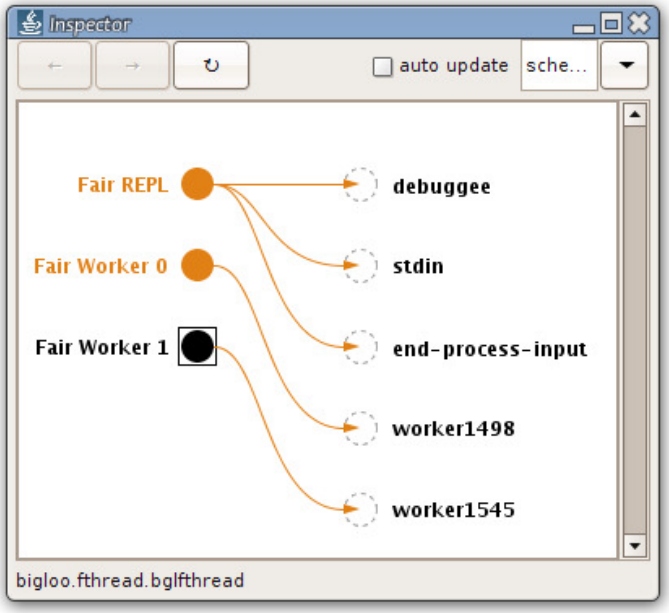


FIG. 8.7: Débogage du débogueur avec l'inspecteur structurel

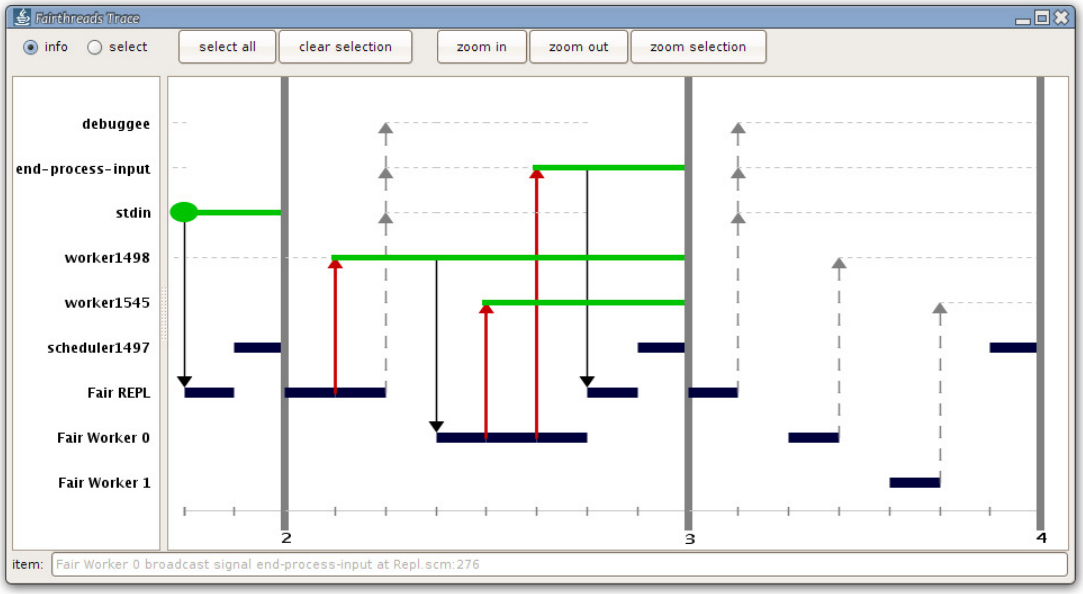


FIG. 8.8: Trace de l'action déficiente dans le débogueur

8.8 Expérience pratique

8.8.1 Implantation dans le débogueur

Les nouvelles fonctionnalités de débogage n'ont nécessité aucune modification de la bibliothèque des *Fair Threads*. L'implantation a été réalisée au moyen d'une série de points d'arrêts spécialement « marqués », d'inspection de vue virtuelle de pile et de code ajouté

dynamiquement dans le programme débogué.

Le module d'extension maintient des points d'arrêt inactifs posés sur des fonctions stratégiques de la bibliothèque des *Fair Threads*. Ces points d'arrêts contiennent une marque spéciale pour pouvoir les différencier de points d'arrêts classiques. De plus, le module d'extension est enregistré en tant qu'écouteur d'événements JVMTI (cf. 3.5.3, page 36). Pour implanter un saut comme l'exécution jusqu'au début d'instant ou la fin d'instant, le débogueur réactive le bon point d'arrêt avant de reprendre l'exécution. Lorsque ce dernier est atteint, l'écouteur du module d'extension identifie le point d'arrêt grâce à sa marque et peut ainsi le désactiver jusqu'à ce qu'il soit de nouveau utilisé.

Les informations affichées dans les inspecteurs sont obtenues en inspectant les structures de données de l'ordonnanceur. Certains sauts doivent donc continuer l'exécution jusqu'à la fin de l'opération logique de manière à observer des changements dans l'ordonnanceur. Ainsi, pour la diffusion de signal, un point d'arrêt est posé sur la fonction `broadcast!` et lorsque cette dernière est atteinte, un saut sortant est effectué, afin de suspendre l'exécution une fois le signal diffusé.

La boîte à outils des *Fair Threads* affiche la liste des *threads* vivants dans le programme débogué. Pour maintenir cette liste à jour, le module d'extension écoute les événements JVMTI de création et de terminaison de *thread*. Dès qu'un de ces événements est reçu, le module enregistre auprès de Bugloo une fonction de rappel qui sera exécutée la prochaine fois que le programme se suspendra et qui mettra à jour l'affichage de la boîte à outils.

Des règles de remplacement ont été mises au point pour masquer les fonctions provenant de la bibliothèque d'exécution des *Fair Threads*. Cela a pour effet de conserver des fonctions utilisateur en sommet de pile, ce qui permet de suivre correctement la progression de l'exécution dans l'environnement de programmation.

A l'instar de l'exécution pas-à-pas, la trace des événements est mise en œuvre à l'aide de points d'arrêts placés dans la bibliothèque des *Fair Threads*. La trace est mise à jour au moment même où un nouvel événement se produit, afin de pouvoir enregistrer la position courante dans le code utilisateur. De plus, l'enregistrement de la trace doit avoir lieu dans le débogueur car il faut construire une vue virtuelle de la pile d'exécution pour que la position courante soit valide. En résumé, l'avantage de ce mécanisme de trace est qu'il ne nécessite aucune modification dans le code de la bibliothèque des *Fair Threads*. L'inconvénient et qu'il nécessite de suspendre et de reprendre l'exécution à chaque fois qu'un nouvel événement se produit, ce qui entraîne des ralentissements inévitables.

8.8.2 Bénéfices et limitations des outils développés

Ce module d'extension présenter dans ce chapitre vient compléter le support que fournit déjà Bugloo pour le débogage des abstractions de haut niveau du langage Bigloo. Sa vertu première est de clarifier ce qui se passe dans l'ordonnanceur durant l'exécution. C'est aussi un bon moyen de s'acclimater à la programmation basée sur les signaux et les instants.

Comme résultat direct, l'utilisation de ces outils a permis de déboguer les mécanismes de communications internes du Bugloo, présentés dans le chapitre 3. Comme résultat plus inattendu, ces outils ont permis de détecter deux bogues dans les premières implantations de la bibliothèque de *Fair Threads*. Le premier entraînait une inversion de l'ordonnancement des *threads* d'un instant à un autre. Le second causait une mauvaise insertion dans l'ordonnanceur des *threads* créés en cours d'exécution.

Certaines informations de débogage sont encore limitées. Par exemple, si la trace est un bon outil pour surveiller les communications entre les *threads*, elle ne dit rien sur les fonctions

exécutées par les *threads* entre les événements enregistrés. L'outil de trace pourrait évoluer en s'inspirant des techniques de visualisation de trace développées dans GThreads [ZS95]. Dans ces travaux, Zhao et Stasko décrivent une trace dans laquelle la durée de vie d'un *thread* est décomposée en segments colorés représentant les différentes fonctions exécutées. Une telle idée serait difficile à appliquer dans BUGLOO si l'on veut tenir compte de la vue virtuelle de la pile, mais elle serait bénéfique pour l'utilisateur.

L'outil de trace est perfectible. Par exemple, un défaut de coopération peut entraîner des exécutions instantanées sans fin. Cela peut faire grossir la trace jusqu'à la rendre trop grosse pour être affichée. Une solution à ce problème serait de s'inspirer de l'outil d'analyse de trace Jinsight [SDPK01] qui permet de détecter des répétitions de séquences d'appels de fonctions et de les regrouper afin d'obtenir une représentation graphique de la trace plus compacte [PLVW98]. Enfin, il serait nécessaire de fournir un moyen de masquer des signaux ou des *threads* pour permettre d'analyser correctement des traces de gros programmes contenant beaucoup de *threads* ou diffusant beaucoup de signaux.

8.9 Conclusion

Ce chapitre a présenté un module d'extension BUGLOO offrant un support de débogage pour les *Fair Threads*, une bibliothèque de programmation concurrente fournie dans le langage Bigloo. Le module est composé de nouvelles vues pour l'inspecteur graphique qui permettent visualiser efficacement l'état de l'ordonnanceur ou des *threads*. De plus, le module propose un outil permettant de tracer le déroulement de l'ordonnancement des *threads* et de visualiser le résultat de manière graphique.

L'implantation de ce module d'extension a consisté en la création et le marquage de points d'arrêt, l'emploie du mécanisme d'écoute d'événements du BUGLOO et l'utilisation de fonctions de rappels. Il n'a pas été nécessaire de modifier la bibliothèque des *Fair Threads*. En conclusion, ce chapitre a montré que le fait que BUGLOO exporte ses mécanismes internes de contrôle du débogué à travers une API permet aux implanteurs de développer aisément leurs propres fonctionnalités de débogage.

Chapitre 9

Conclusion

ette thèse est consacrée à l'amélioration des techniques de débogage symbolique pour tenir compte des spécificités des langages de haut niveau, notamment leur compilation délicate vers des plates-formes d'exécution généralistes. Le résultat de ces travaux est la réalisation de Bugloo, un débogueur multi-langages pour la machine virtuelle Java. Ce débogueur est implanté au moyen de JVMTI, une API standard pour la construction de débogueurs JVM. La plateforme JVM est attrayante du point de vue du débogage, car elle permet à l'utilisateur d'éviter de compiler son programme dans un mode *spécial débogage* et de limiter les chutes de performances lors du débogage.

Bugloo est un débogueur extensible. Il expose une interface de programmation unique aux utilisateurs du débogueur et aux implanteurs de langages. Les premiers peuvent accéder à cet interface à travers la ligne de commande Scheme afin de personnaliser certaines fonctionnalités du débogueur ou pour construire des macro-fonctions. Les implanteurs de langage utilisent l'interface pour spécialiser certaines fonctionnalités du débogueur comme les points d'arrêts (cf. chapitre 6). Ils l'utilisent aussi pour ajouter des fonctionnalités spécifiques aux caractéristiques de leur langage (cf. chapitre 8).

Afin de masquer les détails d'implantation des langages de haut niveau, il est nécessaire de présenter une vue virtuelle de l'état du débogué à l'utilisateur. Chaque implanteur de langage est en charge de fournir au débogueur ses fonctionnalités de décodage des identificateurs, d'affichage des types de données du langage ou encore de recherche de variables locales. Si nécessaire, l'implanteur peut fournir des représentations virtuelles pour les objets structurés *ad-hoc* du langage (cf. 3.4.2, page 33), afin de masquer les détails de leur implantation lors de l'inspection structurelle.

Ces travaux montrent ensuite que pour obtenir un débogage efficace des langages de haut niveau, il est nécessaire de masquer dans la pile les fonctions intermédiaires produites par le compilateur, ainsi que les fonctions de bibliothèques responsables de l'évaluation de code interprété *ad-hoc*. Le débogueur doit aussi s'assurer que l'exécution pas-à-pas ne s'arrête jamais dans des fonctions intermédiaires. Enfin, les sauts de l'exécution pas-à-pas doivent avoir un effet relatif au code source et non au code produit. Il doivent donc fonctionner correctement quel que soit la représentation du code source à l'exécution.

Pour traiter le problème de pollution de pile, la solution développée dans ces travaux s'articule autour de trois nouvelles notions. La première notion, appelée filtre de bloc, permet localiser un bloc d'activation en fonction de ses caractéristiques. La seconde notion

est un langage appelé *Omega* utilisant les filtres de bloc pour identifier des séquences de blocs d'activation. La troisième notion est un algorithme de construction de vue virtuelle de pile. Cet algorithme est guidé par un ensemble de règles de remplacement fournies par les implanteurs de langages. Il permet de remplacer des séquences de blocs d'activation par des blocs virtuels, c'est-à-dire des blocs qui ne sont pas physiquement présent dans la pile. En itérant ce processus sur l'ensemble de la pile, on obtient une représentation virtuelle dans laquelle chaque détail de compilation (appel de fonction ou structure de donnée intermédiaire) a disparu.

Afin de gérer correctement l'exécution pas-à-pas, des techniques de filtrage de sauts permettent de relancer temporairement l'exécution lorsque le sommet de pile contient des fonctions non désirables. Ces situations sont spécifiées par les implanteurs de langage à l'aide du langage *Omega*. Enfin, les implanteurs peuvent redéfinir l'effet des sauts en fonction des blocs d'activation virtuels, ce qui permet de conserver une exécution pas-à-pas correcte, même en présente de code interprété en sommet de pile.

Un chapitre a été consacré au débogage mémoire des langages de haut niveau dont le fonctionnement repose sur la désallocation automatique. Pour lutter contre les fuites mémoires, Bugloo fournit un inspecteur d'allocation capable de déterminer le site d'allocation des objets. De plus, il permet d'inspecter graphiquement les références liant les objets du tas et d'exhiber les racines du GC responsables de rétentions d'objets. A ce jour, il manque un mécanisme permettant de reconstruire une représentation logique des liens existant entre les objets. Par conséquents, les détails de compilation et les structures de données intermédiaires sont toujours visibles dans l'inspecteur.

Un profileur d'allocation mémoire a été spécialement développé dans le but de tenir compte de la compilation délicate des langages de haut niveau. Cet outil fonctionne par instrumentation durant l'exécution, il ne nécessite pas de compiler son programme dans un mode spéciale. Une fois les informations récoltées, le débogueur emploie une technique dérivée de la construction de vue virtuelle afin de conserver uniquement les fonctions utilisateur dans les statistiques finales. Les allocations effectuées par les fonctions intermédiaires sont automatiquement imputées aux fonctions utilisateurs. Dans l'implantation actuelle, la taille des informations récoltées est proportionnelle au nombre d'allocations effectuées, ce qui limite l'utilisation du profileur. De plus, les informations récoltées ne permettent pas encore de construire une représentation virtuelle correcte des fonctions interprétées.

Durant ces travaux, un support de débogage complet a été développé pour le langage Bigloo, un dialecte du langage fonctionnel Scheme. Les résultats ont montré qu'il était possible de masquer tous les détails d'implantation d'un langage de haut-niveau complet. Citons par exemple la compilation des fonctions d'ordre supérieurs, l'implantation de traitant d'exception *ad-hoc* ou encore l'évaluation de fonctions interprétées. Les résultats ont aussi montré qu'une prise en charge totale d'un langage ne nécessite qu'une légère adaptation du compilateur de la part des implanteurs du langage.

Des expérimentations similaires ont étés menées sur Rhino, une implantation d'ECMAScript pour la JVM, ainsi que sur Jython, une implantation de Python. Bien que partielle, la prise en charge de ces langages a permis de montrer que les techniques de représentations virtuelles développées peuvent s'appliquer efficacement quelle que soit la complexité de la compilation d'un langage de haut niveau. Enfin, des tests de débogage ont montré que ces techniques restent efficaces lorsqu'un programme est composé de plusieurs langages de haut

niveau.

Plusieurs pistes sont envisageables pour étendre un peu plus les capacités de débogage des langages de haut niveau. Tout d'abord, il serait souhaitable de mettre au point un mécanisme de représentation virtuelle pour masquer les détails d'implantation durant l'inspection des références entre les objets du tas. En ce qui concerne la représentation virtuelle de pile, il serait intéressant de déterminer si les règles de remplacements pourraient être compilées en une sorte d'automate. Cela éliminerait les calculs superflus effectués par le moteur de recherche avec retour arrière (cf. 4.6.2) lorsque plusieurs règles sont similaires.

Des travaux doivent être poursuivis pour améliorer les techniques de profilage des langages de haut niveau. Le profileur mémoire peut être facilement modifié pour produire des quantités d'informations indépendantes du nombre d'allocations effectuées dans le programme. Il faut poursuivre les recherches afin de déterminer les adaptations à apporter à cette technique pour reconstruire efficacement des statistiques pour les fonctions interprétées. Enfin, les techniques de profilage développées pour l'allocation mémoire pourrait être réutilisées pour profiler le temps d'exécution des fonctions. Cela se révélerait sans doute très utile pour un certain nombre de programmes.

En ce qui concerne le débogueur, il serait intéressant d'étoffer le support de débogage des langages Rhino et Jython. Cet effort passe sans doute par une collaboration avec ces communautés respectives. Il serait intéressant d'entamer de nouvelles expérimentations sur d'autres langages de haut niveaux « en vogue », comme Groovy [Laf04], JRuby [SF04] ou BeanShell [Nie05]. Enfin, il serait intéressant d'intégrer Bugloo dans d'autres environnements programmation, notamment Eclipse [DFK+04], qui dispose d'une interface générique pour l'intégration des débogueurs.

Annexe A

Règles de remplacement pour Bigloo

filter_stack_debuggee.scm


```
(module lang:bigloo:filter-stack-debuggee
   (include "include/common.sch")
   (java
    (class dbgee-ctrl
       (method static bstring->jstring::%jstring (::bstring)
              "bstring_to_jstring")
       (method static symbol->jstring::%jstring (::symbol)
              "symbol_to_jstring")
       "bugloo.control.DebuggeeControl")
    (export stackframe-function-name "getStackFrameFunctionName")
    (export stackframe-function-args "getStackFrameFunctionArgs")
    (export stackframe-file-name "getStackFrameFileName")
    (export stackframe-line "getStackFrameLine")
    (export eval-local-variables "getEvalLocalVariables")
    )
   (export
    (stackframe-function-name::string ::obj)
    (stackframe-function-args::int ::obj)
    (stackframe-file-name::string ::obj)
    (stackframe-line::int ::obj)
    (eval-local-variables::obj ::obj ::obj)
    )
   )

(define (stackframe-function-name bcode)
   (if (vector? bcode)
       (let ((loc (vector-ref bcode 1)))
          (if (and loc (= (length loc) 6))
              (let ((name (list-ref loc 4)))
                 (if (eq? name 'nowhere)
                     "<anonymous>"
                     (symbol->string name)))
              "[code:toplevel-expression]"))
       "[code:sexp]"))

(define (stackframe-function-args bcode)
   (if (vector? bcode)
```

```
38         (let ((loc (vector-ref bcode 1)))
39            (if (and loc (= (length loc) 6))
40                (list-ref loc 5)
41                0))
42         0))
43
44 (define (stackframe-file-name bcode)
45    (let ((loc (vector-ref bcode 1)))
46       (if loc
47           (cadr loc)
48           "???")))
49
50 (define (stackframe-line bcode)
51    (let ((loc (vector-ref bcode 1)))
52       (if loc
53           (let ((x (cadddr loc)))
54              (if (integer? x)
55                  x
56                  -1))
57           -1)))
58
59 (define (eval-local-variables stack names)
60    (vector
61     (list->vector stack)
62     (list->vector
63      (map (lambda (x) (dbgee-ctrl-symbol->jstring x)) names)) ))
64
```

filter_stack.scm

```
1  (module lang:bigloo:filter-stack
2     (import
3      features:mangler
4      features:localvar
5      features:stackframe
6      filters:stack
7      )
8
9     (include "_filter_stack.sch")
10
11    (export
12     ;;Specializedstackframestypesforbigloo
13     (class closure-stackframe::stackframe)
14     (class eval-stackframe::stackframe)
15     (class eval-macro-stackframe::eval-stackframe)
16     (class generic-stackframe::stackframe)
17     (create-closure-stackframe ::stackframe)
18     (create-eval-stackframe ::stackframe)
19     (create-eval-macro-stackframe ::stackframe)
20     (create-generic-stackframe ::stackframe ::stackframe)
21     ;;Specializedvariables
22     (class captured-localvar::localvar)
23     (create-captured-localvar name env index)
24     )
25    )
26
27 ;*---------------------------------------------------------------------*/
28 ;*    stackframe specialization functions. Implementation and          */
29 ;*    installation are done in the backend source file...              */
30 ;*---------------------------------------------------------------------*/
31 (define-method (stackframe-init o::closure-stackframe)
32    (call-next-method))
33
34 (define-method (stackframe-init o::eval-stackframe)
35    (call-next-method))
36
```

```
;*---------------------------------------------------------------------*/
;*    Install the specilized stackframe in the debugger...             */
;*---------------------------------------------------------------------*/
(stack-filter-add-transform-function!
 'sf-bigloo-with-output
 (lambda (frames) (list (cadr frames))))

(stack-filter-add-transform-function!
 'sf-bigloo-high-order-procedure-call
 (lambda (frames) (list (car frames))))

(stack-filter-add-transform-function!
 'sf-bigloo-high-order-closure-call
 (lambda (frames) (list (create-closure-stackframe (car frames)))))

(stack-filter-add-transform-function!
 'sf-bigloo-generic-call
 (lambda (f)
    (list (create-generic-stackframe (car f) (caddr f)))))

(stack-filter-add-pending-function!
 'rrp-bigloo-try
 (lambda (frames)
    (let ((exc-frame (car frames)))
       (lambda (f)
          (stackframe-line-set! f (delay (stackframe-line exc-frame)))
          (stackframe-%line/stratum-set!
           f (lambda (sf stratum)
                (get-stackframe-line/stratum exc-frame stratum)))
          f))))


(define (last-evmeaning lf)
   (if (and (pair? (cdr lf))
            (string=? (stackframe-function-name (cadr lf)) "evmeaning"))
       (last-evmeaning (cdr lf))
       (car lf)))

(stack-filter-add-transform-function!
 'sf-bigloo-evmeanings
 (lambda (frames)
    (let* ((lsf (create-eval-stackframe (last-evmeaning frames)))
           (macro? (eval-stackframe-is-macro? lsf)))
       (if macro? (set! lsf (create-eval-macro-stackframe lsf)))
       (if (pair? (cdr frames))
           (let ((fsf (create-eval-stackframe (car frames))))
              (if macro? (set! fsf (create-eval-macro-stackframe fsf)))
              (stackframe-line-set! lsf (delay (stackframe-line fsf)))
              (stackframe-local-variables-set!
               lsf (delay (stackframe-local-variables fsf)) )))

       (list lsf) )))

(stack-filter-add-pending-function!
 'rrp-bigloo-evmeaning-prep-call
 (lambda (frames)
    (let ((fsf (create-eval-stackframe (car frames))))
       (lambda (f)
          (if (eval-macro-stackframe? f)
              (set! fsf (create-eval-macro-stackframe fsf)))
          (stackframe-line-set! f (delay (stackframe-line fsf)))
          (stackframe-local-variables-set!
           f (delay (stackframe-local-variables fsf)))
          f))))

(define (mem-ref mem i)
   (list-ref mem (-fx i 1)))

(stack-filter-add-regexp-function!
 're-bigloo-args
 (lambda (args mem)
    (let* ((l (length args))
           (lm (length (mem-ref mem 3)))
           (b (or (=fx lm 1)
```

```
                  (and (=fx l lm);(length (mem-ref mem 3)))
                       (every?
                        (lambda (x)
                           (string=? x "java.lang.Object")) args))))
           (str (cond
                  ((=fx lm 1) ; bigloo optional arguments
                   "funcall")
                  ((<=fx l 4)
                   (string-append "funcall" (integer->string l)))
                  (else
                   "apply"))))
       (cons b (if b (append mem (list str)) mem)))))

(stack-filter-add-regexp-function!
 're-bigloo-args-next
 (lambda (args mem)
    (let* ((l (length args))
           (b (every? (lambda (x) (string=? x "java.lang.Object")) args))
           (str (cond
                  ((=fx l 1)
                   "funcall")
                  ((<=fx l 4)
                   (string-append "funcall" (integer->string l)))
                  (else
                   "apply"))))
       (cons b (if b (append mem (list str)) mem)))))


(stack-filter-add-regexp-function!
 're-bigloo-gen-name
 (lambda (name mem)
    (let* ((mgl (demangle-ident name))
           (re (pregexp-match ".*_([0-9]*)_[0-9]*" mgl)))
       (if re
           (let* ((pos (string->integer (cadr re)))
                  (funname (substring mgl 0 pos)))
              (cons #t (append mem (list funname))))
           (cons #f mem)))))
```

_filter_stack.scm

```
;*---------------------------------------------------------------------*/
;*    The implementation of the specialized stackframes                */
;*---------------------------------------------------------------------*/
(define (create-closure-stackframe orig::stackframe)
   (let ((opeer (impl-stackframe-%peer (stackframe-%impl orig))))
      (instantiate::closure-stackframe
         (%impl (instantiate::impl-closure-stackframe
                   (i-orig (stackframe-%impl orig))
                   (%peer (%jvm-closure-stackframe-new opeer)))))))

(define-method (%stackframe-function-name::string
                sf::impl-closure-stackframe)
   (let ((str (%stackframe-function-name
               (impl-closure-stackframe-i-orig sf))))
      (cond
         ((substring-at? str "BgL__" 0)
          (string-append "BgL_" (substring str 5 (string-length str))))
         ((substring-at? str "_" 0)
          (substring str 1 (string-length str)))
         (else
          str))))
```

```
(define-method (%stackframe-function-args-type::pair-nil
                sf::impl-closure-stackframe)
   (cdr (%stackframe-function-args-type
         (impl-closure-stackframe-i-orig sf))) )


(define-method (%stackframe-function-args-count::int
                sf::impl-closure-stackframe)
   (%stackframe-function-args-count
    (impl-closure-stackframe-i-orig sf)))

(define (create-captured-localvar name env index)
   (instantiate::captured-localvar
      (%impl (instantiate::impl-captured-localvar
                (%peer (%jvm-captured-localvar-new name env index))))))

(define (make-captured-localvars dbg-bigloo-proc off names)
   (let loop ((off off)
              (i 0)
              (index 0)
              (res '()))
      (if (pair? off)
          (let ((x (string->integer (car off))))
             (loop
              (cdr off)
              x
              (+fx 1 index)
              (cons (create-captured-localvar
                     (substring names i (+ i x))
                     dbg-bigloo-proc
                     index)
                    res)))
          (reverse res))))

(define-method (%stackframe-local-variables sf::impl-closure-stackframe)
   (let ((l (call-next-method)))
      (if (pair? l)
          (let ((e (localvar-name (car l))))
             (if (substring-at? e "<env:" 0)
                 (let* ((s (string-split e ":"))
                        (proc (localvar-value (car l)))
                        (off (string-split (cadr s) "_"))
                        (names (caddr s))
                        (vars (make-captured-localvars proc off names)))
                   (append vars (cdr l)) )
                 l))
          '() )))



;*---------------------------------------------------------------------*/
;*    Stackframe specialization for evaluator                          */
;*---------------------------------------------------------------------*/
(define (create-eval-stackframe orig::stackframe)
   (let ((opeer (impl-stackframe-%peer (stackframe-%impl orig))))
      (instantiate::eval-stackframe
         (%impl (instantiate::impl-eval-stackframe
                   (%peer (%jvm-eval-stackframe-new opeer)))))))


(define-method (%stackframe-filename sf::impl-eval-stackframe)
   (%jvm-eval-stackframe-filename
    (impl-eval-stackframe-%peer sf)) )

(define-method (%stackframe-filename/stratum sf::impl-eval-stackframe
                                             stratum)
   (%jvm-eval-stackframe-filename
    (impl-eval-stackframe-%peer sf)) )

(define-method (%stackframe-line sf::impl-eval-stackframe)
   (%jvm-eval-stackframe-line
    (impl-eval-stackframe-%peer sf)) )

(define-method (%stackframe-line/stratum sf::impl-eval-stackframe
```

```
                                        stratum)
   (%jvm-eval-stackframe-line
    (impl-eval-stackframe-%peer sf)) )



;*---------------------------------------------------------------------*/
;*    Stackframe specialization for macro (define-macro)               */
;*---------------------------------------------------------------------*/
(define (eval-stackframe-is-macro? orig::eval-stackframe)
   (let ((x (%jvm-eval-stackframe-is-macro?
             (impl-stackframe-%peer (stackframe-%impl orig)))))
      x))

(define (create-eval-macro-stackframe orig::stackframe)
   (let ((opeer (impl-stackframe-%peer (stackframe-%impl orig))))
      (instantiate::eval-macro-stackframe
         (%impl (instantiate::impl-eval-macro-stackframe
                   (%peer (%jvm-eval-stackframe-new opeer)))))))

(define macrofilter (pregexp "^(\\.dummy\\.[0-9]*|.*\\.defmacro)$"))
(define-method (%stackframe-local-variables sf::impl-eval-macro-stackframe)
   (let ((x (call-next-method)))
      (filter (lambda (x)
                 (not (pregexp-match macrofilter (localvar-name x)))) x)
))


;*---------------------------------------------------------------------*/
;*    Stackframe specialization for generic methods...                 */
;*---------------------------------------------------------------------*/
(define (create-generic-stackframe orig::stackframe types::stackframe)
   ;(print "create-stackframe: " orig " - " types)
   (let* ((opeer (impl-stackframe-%peer (stackframe-%impl orig)))
          (on (demangle-ident (stackframe-function-name orig)))
          (tn (demangle-ident (stackframe-function-name types)))
          (p (caadr (pregexp-match-positions ".*(_[0-9]*_)[0-9]*" on))))
      (instantiate::generic-stackframe
         (%impl (instantiate::impl-generic-stackframe
                   (gentype (substring on (+fx (string-length tn) 1) p))
                   (typessf (stackframe-%impl types))
                   (%peer (%jvm-generic-stackframe-new opeer)))))))

(define-method (%stackframe-function-name sf::impl-generic-stackframe)
   (%stackframe-function-name (impl-generic-stackframe-typessf sf)))

(define-method (%stackframe-function-args-type sf::impl-generic-stackframe)
   (cons (impl-generic-stackframe-gentype sf)
         (cdr (%stackframe-function-args-type
               (impl-generic-stackframe-typessf sf)))))

(define-method (%stackframe-function-args-count sf::impl-generic-stackframe)
   (%stackframe-function-args-count
               (impl-generic-stackframe-typessf sf)))
```

_filter_stack.sch

```
(directives
   (java
    ;;TODO:shouldberemoved
    (class %jvm-closure-stackframe
       (constructor new (::%jvm-stackframe))
       (method connection-set!::void (::%jvm-closure-stackframe ::obj)
               "setConnection")
       (method pos::int (::%jvm-closure-stackframe)
```

```
            "getPosition")
      (method function-name::string (::%jvm-closure-stackframe)
            "getFunctionName")
      (method function-ret-type::string (::%jvm-closure-stackframe)
            "getFunctionReturnType")
      (method function-args-type::pair-nil (::%jvm-closure-stackframe)
            "getFunctionArgumentsType")
      (method function-args-count::int (::%jvm-closure-stackframe)
            "getFunctionArgumentCount")
      (method function-class::string (::%jvm-closure-stackframe)
            "getFunctionClass")
      (method filename::string (::%jvm-closure-stackframe)
            "getFilename")
      (method line::int (::%jvm-closure-stackframe)
            "getLine")
     "bugloo.lang.bigloo.ClosureStackFrameImpl")
   (class %jvm-eval-stackframe
      (constructor new (::%jvm-stackframe))
      (method connection-set!::void (::%jvm-eval-stackframe ::obj)
            "setConnection")
      (method pos::int (::%jvm-eval-stackframe)
            "getPosition")
      (method function-name::string (::%jvm-eval-stackframe)
            "getFunctionName")
      (method function-ret-type::string (::%jvm-eval-stackframe)
            "getFunctionReturnType")
      (method function-args-type::pair-nil (::%jvm-eval-stackframe)
            "getFunctionArgumentsType")
      (method function-args-count::int (::%jvm-eval-stackframe)
            "getFunctionArgumentCount")
      (method function-class::string (::%jvm-eval-stackframe)
            "getFunctionClass")
      (method filename::string (::%jvm-eval-stackframe)
            "getFilename")
      (method line::int (::%jvm-eval-stackframe)
            "getLine")
      (method get-value-for-prop::obj (::%jvm-eval-stackframe
                                       ::obj)
            "getValueForProp")
      (method set-props/values!::void (::%jvm-eval-stackframe
                                       ::pair-nil)
            "setPropsWithValues")
      (method is-macro?::bool (::%jvm-eval-stackframe)
            "isAMacro")
      "bugloo.lang.bigloo.EvalStackFrameImpl")
   (class %jvm-generic-stackframe
      (constructor new (::%jvm-stackframe))
      "bugloo.lang.bigloo.GenericStackFrameImpl")

   (class %jvm-captured-localvar
      (constructor new (::string ::%jdi-objectreference ::int))
      "bugloo.lang.bigloo.CapturedLocalVarImpl")
   )

  (import
   features:localvar
   )

  (export
   (class impl-closure-stackframe::impl-stackframe
      i-orig)
   (class impl-eval-stackframe::impl-stackframe)
   (class impl-eval-macro-stackframe::impl-eval-stackframe)
   (class impl-generic-stackframe::impl-stackframe
      gentype
      typessf)

   (class impl-captured-localvar::impl-localvar)
   )

  )
```

GenericStackFrameImpl.java

```
1  package bugloo.lang.bigloo;
2
3  import java.util.*;
4
5  import com.sun.jdi.*;
6  import com.sun.jdi.request.*;
7
8  import bigloo.*;
9
10 import bugloo.features.*;
11
12 public class GenericStackFrameImpl extends StackFrameImpl {
13     public GenericStackFrameImpl() {
14     }
15
16     public GenericStackFrameImpl(StackFrameImpl sfi) {
17         super(sfi.getStackFrame(), sfi.pos);
18     }
19 }
```

EvalStackFrameImpl.java

```
1  package bugloo.lang.bigloo;
2
3  import java.util.*;
4
5  import com.sun.jdi.*;
6  import com.sun.jdi.request.*;
7
8  import bigloo.*;
9
10 import bugloo.control.*;
11 import bugloo.features.*;
12
13 public class EvalStackFrameImpl extends StackFrameImpl {
14
15     public static void setDebuggeeSessionParams() {
16         new CallbackAction(
17               Action.conf.program,
18               new Runnable() {
19                   public void run() {
20                       Action.loadClassIntoDebuggee(
21                          "bugloo.lang.bigloo.BglStackFiltersDebuggee");
22                   }
23               });
24     }
25
26     public EvalStackFrameImpl() {
27     }
28
29     public EvalStackFrameImpl(StackFrameImpl sfi) {
30         super(sfi.getStackFrame(), sfi.pos);
31     }
32
33     public byte[] getFunctionName() {
34         try {
35             StackFrame sf=getStackFrame();
```

```
            LocalVariable v=sf.visibleVariableByName("code");
            Value code=sf.getValue(v);
            LinkedList l=new LinkedList();
            l.add(code);
            return ((StringReference)Action.
             invokeRemoteMethod(
                       "bugloo.lang.bigloo.BglStackFiltersDebuggee",
                       "getStackFrameFunctionName",l))
                .value().getBytes();
        } catch (AbsentInformationException e) {
            return "[code:s-exp]".getBytes();
        }
    }

    public Object getFunctionArgumentsType() {
        Object ret=nil.nil;
        int arg=0;
        try {
            Value code=getStackFrame()
                .getValue(getStackFrame().visibleVariableByName("code"));
            LinkedList l=new LinkedList();
            l.add(code);
            arg=((IntegerValue)Action
                 .invokeRemoteMethod(
                   "bugloo.lang.bigloo.BglStackFiltersDebuggee",
                   "getStackFrameFunctionArgs",l)).value();
        } catch (AbsentInformationException e) {
            e.printStackTrace();
        }
        if (arg<0) {
            arg=(-arg)-1;
            ret=new pair("obj".getBytes(),ret);
            ret=new pair(".".getBytes(),ret);
        }
        while (arg-->0) {
            ret=new pair("obj".getBytes(),ret);
        }
        return ret;
    }

    public byte[] getFilename() {
        try {
            Value code=getStackFrame()
                .getValue(getStackFrame().visibleVariableByName("code"));
            LinkedList l=new LinkedList();
            l.add(code);
            return ((StringReference)Action
               .invokeRemoteMethod(
                       "bugloo.lang.bigloo.BglStackFiltersDebuggee",
                       "getStackFrameFileName",l)).value().getBytes();
        } catch (AbsentInformationException e) {
            return "???".getBytes();
        }
    }

    public int getLine() {
        if (line_set) {
            return line;
        } else {
            try {
                Value code=getStackFrame()
                    .getValue(getStackFrame()
                              .visibleVariableByName("code"));
                LinkedList l=new LinkedList();
                l.add(code);
                return ((IntegerValue)Action
                  .invokeRemoteMethod(
                   "bugloo.lang.bigloo.BglStackFiltersDebuggee",
                   "getStackFrameLine",l)).value();
            } catch (AbsentInformationException e) {
                return -1;
            }
        }
```

```
109     }
110
111     public Object getLocalVariables() {
112         pair cons=new pair(nil.nil,nil.nil);
113         Object tmp=cons;
114         try {
115             LinkedList args=new LinkedList();
116             StackFrame sf=getStackFrame();
117             args.add(sf.getValue(sf.visibleVariableByName("stack")));
118             args.add(sf.getValue(sf.visibleVariableByName("names")));
119             Value v=Action.invokeRemoteMethod(
120                         "bugloo.lang.bigloo.BglStackFiltersDebuggee",
121                         "getEvalLocalVariables",args);
122             List l=((ArrayReference)v).getValues();
123             Iterator s=((ArrayReference)l.get(0)).getValues().iterator();
124             Iterator n=((ArrayReference)l.get(1)).getValues().iterator();
125             while (s.hasNext()) {
126                 Value val=(Value)s.next();
127                 Value name=(Value)n.next();
128                 String str=((StringReference)name).value();
129                 //System.out.println(""+name+","+val);
130                 ((pair)tmp).cdr=
131                     new pair (new EvalLocalVarImpl(str,val), nil.nil);
132                 tmp=((pair)tmp).cdr;
133             }
134         } catch (AbsentInformationException e) {
135             e.printStackTrace();
136         }
137         return cons.cdr;
138     }
139
140     public boolean isAMacro() {
141         try {
142             StackFrame sf=getStackFrame();
143             ArrayReference ar=(ArrayReference)sf
144                 .getValue(sf.visibleVariableByName("code"));
145             ObjectReference loc=(ObjectReference)ar.getValue(1);
146             ReferenceType n=(ReferenceType)
147                 (Action.conf.vm.classesByName("bigloo.nil").get(0));
148             ReferenceType p=(ReferenceType)
149                 (Action.conf.vm.classesByName("bigloo.pair").get(0));
150             Field f=p.fieldByName("cdr");
151             int i=0;
152             while (!(loc.type().equals(n))) {
153                 i++;
154                 //System.out.println(loc);
155                 loc=(ObjectReference)(loc.getValue(f));
156             }
157             return i==6;
158         } catch (AbsentInformationException e) {
159             e.printStackTrace();
160             return false;
161         }
162     }
163 }
```

EvalLocalVarImpl.java

```
1 package bugloo.lang.bigloo;
2
3 import com.sun.jdi.*;
4 import com.sun.jdi.request.*;
5
6 import java.io.*;
```

```
import java.util.*;

import bigloo.*;

import bugloo.control.*;
import bugloo.features.*;

public class EvalLocalVarImpl implements LocalVar {
    public String name;
    public Value value;
    public NameMangler mgl;
    public Displayer d;

    public EvalLocalVarImpl(String name, Value value) {
        this.name=name;
        this.value=value;
        mgl=ManglingManager.getDemanglerForType(value.type());
        d=DisplayingManager.getCurrentDisplayerForMangler(mgl);
    }

    public byte[] getName() {
        return getName(false);
    }

    public byte[] getName(boolean mustTag) {
        if (mustTag)
            return ("<ident>"+name+"</ident>").getBytes();
        else
            return name.getBytes();
    }

    public byte[] getTypeName() {
        return getTypeName(false);
    }

    public byte[] getTypeName(boolean mustTag) {
        return mgl.demangleType(value.type(),mustTag).getBytes();
    }

    public Object getValue() {
        return value;
    }

    public Displayer getDisplayerForVariableValue() {
        return d;
    }
}
```

ClosureStackFrameImpl.java

```
package bugloo.lang.bigloo;

import java.util.*;

import com.sun.jdi.*;
import com.sun.jdi.request.*;

import bigloo.*;

import bugloo.features.*;

public class ClosureStackFrameImpl extends StackFrameImpl {
    public ClosureStackFrameImpl() {
    }

    public ClosureStackFrameImpl(StackFrameImpl sfi) {
```

```
        super(sfi.getStackFrame(), sfi.pos);
    }

    public byte[] getFunctionName() {
        String r=m.name();
        if (r.charAt(0)=='_') r=r.substring(1);
        return r.getBytes();
    }

    public Object getFunctionArgumentsType() {
        Iterator i=m.argumentTypeNames().iterator();
        pair cons=new pair(nil.nil,nil.nil);
        Object tmp=cons;
        //Weignorethefirstargumentofthefunction,
        //sinceit'stheenvirommentoftheclosure.
        i.next();
        while (i.hasNext()) {
            ((pair)tmp).cdr=new pair (((String)i.next()).getBytes(),nil.nil);
            tmp=((pair)tmp).cdr;
        }
        return cons.cdr;
    }


}
```

CapturedLocalVarImpl.java

```
package bugloo.lang.bigloo;

import com.sun.jdi.*;
import com.sun.jdi.request.*;

import java.io.*;
import java.util.*;

import bigloo.*;

import bugloo.control.*;
import bugloo.features.*;

public class CapturedLocalVarImpl implements LocalVar {
    public String name;
    public ObjectReference proc_obj;
    public int index;
    public ObjectReference captured_var_value;
    public NameMangler mgl;
    public Displayer d;

    public CapturedLocalVarImpl(byte[] name, ObjectReference proc_obj,
                                int index) {
        this.name=new String(name);
        this.proc_obj=proc_obj;
        this.index=index;
        ReferenceType rt=(ReferenceType)proc_obj.type();
        ArrayReference ar=(ArrayReference)
            proc_obj.getValue(rt.fieldByName("env"));
        captured_var_value=(ObjectReference)ar.getValue(index);
        mgl=ManglingManager.getDemanglerForType(captured_var_value.type());
        d=DisplayingManager.getCurrentDisplayerForMangler(mgl);
    }

    public byte[] getName() {
        return getName(false);
    }

```

```
39      public byte[] getName(boolean mustTag) {
40          if (mustTag)
41              return ("<ident>"+name+"</ident>").getBytes();
42          else
43              return name.getBytes();
44      }
45
46      public byte[] getTypeName() {
47          return getTypeName(false);
48      }
49
50      public byte[] getTypeName(boolean mustTag) {
51          return mgl.demangleType(captured_var_value.type(),mustTag).getBytes();
52      }
53
54      public Object getValue() {
55          return captured_var_value;
56      }
57
58      public Displayer getDisplayerForVariableValue() {
59          return d;
60      }
61  }
```

Bibliographie

[ACN+01] Alpern (B.), Choi (J.-D.), Ngo (T.) *et al.*, « A perturbation-free replay platform for cross-optimized multithreaded applications », dans *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, p. 23, Washington, DC, USA, 2001. IEEE Computer Society.

[AFG+00] Arnold (M.), Fink (S.), Grove (D.) *et al.*, « Adaptive optimization in the Jalapeño JVM », *ACM SIGPLAN Notices*, vol. 35, nº 10, 2000, p. 47–65.

[Aho90] Aho (A. V.), « Algorithms for finding patterns in strings », dans Leeuwen (J.), éditeur, *Handbook of Theoretical Computer Science: Volume A: Algorithms and Complexity*, p. 255–300. Elsevier, Amsterdam, 1990.

[Ang79] Angluin (D.), « Finding patterns common to a set of strings (extended abstract) », dans *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, p. 130–141, Atlanta, Georgia, United States, avril 1979. ACM Press.

[Arm97] Armstrong (J.), « The development of erlang », dans *ICFP'97: proceedings of the International Conference on Functional Programming*, p. 196–203, Amsterdam, The Netherlands, juin 1997. ACM Press.

[Bak95] Baker (H. G.), « CONS should not CONS its arguments, part II: Cheney on the M.T.A. », *ACM SIGPLAN Notices*, vol. 30, nº 9, 1995, p. 17–20.

[Bas85] Baskerville (D. B.), « Graphic presentation of data structures in the DBX debugger ». Rapport technique, University of California at Berkeley, Berkeley, CA, USA, 1985.

[Bas86] Baskerville (D. B.), « Graphic presentation of data structures in the dbx debugger ». Rapport technique nº UCB/CSD-86-260, EECS Department, University of California, Berkeley, 1986.

[BDG+88] Bobrow (D.), DeMichiel (L.), Gabriel (R.) *et al.*, « Common Llisp Object System specification », *ACM SIGPLAN Notices*, nº 23, septembre 1988.

[BKO+02] Boswell (D.), King (B.), Oeschger (I.) *et al.*, *Creating Applications with Mozilla*, chap. Development Tools, p. 474. O'Reilly & Associates, Inc., 2002.

[Boe81] Boehm (B. W.), *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Bou93a] Bourdoncle (F.), « Abstract debugging of higher-order imperative languages », dans *PLDI '93: proceedings of the conference on Programming Language Design and Implementation*, p. 46–55, Albuquerque, New Mexico, USA, 1993. ACM Press.

[Bou93b] Bourdoncle (F.), « Assertion-based debugging of imperative programs by abstract interpretation », dans Sommerville (I.) et Paul (M.), éditeurs, *Proceedings of the 4th European Software Engineering Conference*, p. 501–516, Berlin, Germany, 1993. Springer-Verlag.

[Bou04] Bourne (S.), « A conversation with bruce lindsay. », *ACM Queue*, vol. 2, n° 8, 2004, p. 22–33.

[Car97] Carlsson (R.), « Towards a deadlock analysis for Erlang programs ». Rapport de DÉA, Uppsala University, avril 1997.

[CC77] Cousot (P.) et Cousot (R.), « Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints », dans *Symposium on Principles Of Programming Languages*, p. 238–252, Los Angeles, CA, USA, janvier 1977. ACM Press.

[CDH84] Chailloux (J.), Devin (M.) et Hullot (J.-M.), « LELISP, a portable and efficient LISP system », dans *LFP '84: Proceedings of the ACM Symposium on LISP and Functional Programming*, p. 113–122, Austin, Texas, United States, 1984. ACM Press.

[CFF01] Clements (J.), Flatt (M.) et Felleisen (M.), « Modeling an algebraic stepper », dans *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, vol. 2028 (coll. *Lecture Notes in Computer Science*), p. 320–334, Genova, Italy, avril 2001. Springer-Verlag.

[CH04] Christiansen (J.) et Huch (F.), « Searching for deadlocks while debugging concurrent haskell programs », dans *ICFP'02: proceedings of the International Conference on Functional Programming*, p. 28–39, Snow Bird, UT, USA, octobre 2004. ACM Press.

[Cia03] Ciabrini (D.), « Bugloo: A source level debugger for Scheme programs compiled into JVM bytecode », dans *ILC'03: Proceedings of the International Lisp Conference*, New York City, NY, USA, octobre 2003.

[CK03] Cooper (G.) et Krishnamurthi (S.), « Frtime: Distributed and asynchronous functional reactive programming ». Rapport technique n° CS-03-20, Department of Computer Science, Brown University, 2003.

[Cli98] Clinger (W. D.), « Proper tail recursion and space efficiency », dans *PLDI '98: Proceedings of the conference on Programming Language Design and Implementation*, p. 174–185, Montreal, Quebec, Canada, juin 1998. ACM Press.

[CMM+97] Cohn (M.), Morgan (B.), Morrison (M.) *et al.*, *Java Developer's Reference*. Sams, 1997.

[CMR88] Coutant (D. S.), Meloy (S.) et Ruscetta (M.), « DOC: a practical approach to source-level debugging of globally optimized code », *ACM SIGPLAN Notices*, vol. 23, n° 7, 1988, p. 125–134.

[Com95] Committee (T.). « DWARF debugging information format specification version 2.0 », mai 1995.

[CSL04] Cantrill (B. M.), Shapiro (M. W.) et Leventhal (A. H.), « Dynamic instrumentation of production systems », dans *Proceedings of USENIX 2004 Annual Technical Conference*, p. 15–28, Boston, Ma, juin 2004.

[CUL89] Chambers (C.), Ungar (D.) et Lee (E.), « An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes », dans *OOPSLA '89: Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*, p. 49–70, New Orleans, Louisiana, United States, octobre 1989. ACM Press.

[CvM93] Crocker (R.) et von Mayrhauser (A.), « Maintenance support needs for object-oriented software », dans *Computer Software and Applications Conference (COMPSAC)*, Phoenix, AZ, USA, novembre 1993.

[DFH86] Dybvig (R. K.), Friedman (D. P.) et Haynes (C. T.), « Expansion-passing style: beyond conventional macros », dans *LFP '86: Proceedings of the conference on LISP and Functional Programming*, p. 143–150, Cambridge, Massachusetts, United States, 1986. ACM Press.

[DFK+04] D'Anjou (J.), Fairbrother (S.), Kehn (D.) *et al.*, *Java™ Developer's Guide to Eclipse, 2nd Edition*. Addison Wesley Professional, octobre 2004.

[Dyb98] Dybvig (R. K.), *Chez Scheme User's Guide*. Cadence Research Systems, 1998.

[ECM99] ECMA, *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, édition third, décembre 1999.

[EGH94] Emami (M.), Ghiya (R.) et Hendren (L. J.), « Context-sensitive interprocedural points-to analysis in the presence of function pointers », dans *PLDI '01: Conference on Programming Language Design and Implementation*, p. 242–256, Orlando, Florida, United States, juin 1994. ACM Press.

[EJ03] Ennals (R.) et Jones (S. P.), « Hsdebug: debugging lazy programs by not being lazy », dans *Haskell '03: Proceedings of the workshop on Haskell*, p. 84–87, Uppsala, Sweden, 2003. ACM Press.

[Esp04] Esposito (D.), *Introducing Microsoft® ASP.NET 2.0*. Microsoft Press, juillet 2004.

[Et04] EJ-technologies. « Jprofiler overview ». http://www.ej-technologies.com/products/jprofiler/overview.html, 2004.

[FCCF+02] Findler (R. B.), Clements (J.), Cormac Flanagan (M. F.) *et al.*, « DrScheme: A progamming environment for Scheme », *Journal of Functional Programming*, vol. 12, n° 2, mars 2002, p. 159–182.

[Fee00] Feeley (M.). « Scheme Request For Implementation 18: Multithreading support ». http://srfi.schemers.org/srfi-18/srfi-18.html, 2000.

[FF96] Flanagan (C.) et Felleisen (M.), « Modular and polymorphic set-based analysis: Theory and practice ». Rapport technique n° TR96-266, Rice University, 1996.

[FF00a] Flanagan (C.) et Freund (S. N.), « Type-based race detection for java », dans *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, p. 219–232, Vancouver, British Columbia, Canada, juin 2000. ACM Press.

[FF00b] Flanagan (C.) et Freund (S. N.), « Type-based race detection for Java. », dans *PLDI '00: Conference on Programming Language Design and Implementation*, p. 219–232, Vancouver, British Columbia, Canada, juin 2000. ACM Press.

[Fie99] Fiedler (N.). « JSWAT, the java debugger ». http://www.bluemarsh.com/java/jswat/index.html, 1999.

[Fie03] Field (R.), « JSR 45: Debugging support for other languages. ». Rapport technique, Sun Microsystems Inc., http://www.jcp.org/en/jsr/detail?id=45, novembre 2003.

[Fou98] Foundation (M.). « Rhino: Javascript for Java », 1998. http://www.mozilla.org/rhino.

[GG01] Gough (J. J.) et Gough (K. J.), *Compiling for the .NET Common Language Runtime*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[GKM82] Graham (S. L.), Kessler (P. B.) et McKusick (M. K.), « gprof: a call graph execution profiler », dans *SIGPLAN Symposium on Compiler Construction*, p. 120–126, Boston, MA, juin 1982.

[GKNV93] Gansner (E. R.), Koutsofios (E.), North (S. C.) et Vo (K.-P.), « A technique for drawing directed graphs », *IEEE Transactions on Software Engineering*, vol. 19, n° 3, 1993, p. 214–230.

[GLS78] Guy L. Steele (J.), « Rabbit: A compiler for scheme ». Rapport technique n° GIT-GVU-95-01, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[GR83] Goldberg (A.) et Robson (D.), *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[GS03a] Gallesio (E.) et Serrano (M.), « Programming graphical user interfaces with Scheme », *Journal of Functional Programming*, vol. 13, n° 5, septembre 2003, p. 839–866.

[GS03b] Gallesio (E.) et Serrano (M.), « Programming graphical user interfaces with Scheme », *Journal of Functional Programming*, vol. 13, n° 5, 2003, p. 839–866.

[Han91] Hanson (C.), « MIT Scheme reference manual ». Rapport technique n° AITR-1281, Massachusetts Institute of Technology, Cambridge, MA, USA, 1991.

[HCU92] Hölzle (U.), Chambers (C.) et Ungar (D.), « Debugging optimized code with dynamic deoptimization », dans *PLDI '92: Proceedings of the conference on Programming Language Design and Implementation*, p. 32–43, San Francisco, California, United States, juin 1992. ACM Press.

[HDD06] Hofer (C.), Denker (M.) et Ducasse (S.), « Design and implementation of a backward-in-time debugger », dans *Proceedings of NODE'06*, 2006.

[Hin01] Hind (M.), « Pointer analysis: haven't we solved this problem yet? », dans *PASTE '01: Proceedings of the workshop on Program analysis for software tools and engineering*, p. 54–61, Snowbird, Utah, United States, 2001. ACM Press.

[HJ91] Hastings (R.) et Joyce (B.), « Purify: Fast detection of memory leaks and access errors », dans *Proceedings of the 1992 USENIX Conference*, p. 125–138, San Francisco, CA, USA, 1991.

[HJ94] Heintze (N.) et Jaffar (J.), « Set constraints and set-based analysis », dans *PPCP'94: proceedings of the 2nd international workshop on Principles and*

Practice of Constraint Programming, vol. 874 (coll. *Lecture Notes in Computer Science*), p. 281–298, Berlin, Germany, 1994. Springer-Verlag.

[How95] Howard (T. G.), *Smalltalk Developer's Guide to VisualWorks*. Cambridge University Press, New York, NY, USA, 1995.

[HP98] Havelund (K.) et Pressburger (T.), « Model checking Java programs using Java PathFinder », *Software Tools for Technology Transfer*, vol. 2, n° 4, avril 1998.

[Hug97] Hugunin (J.), « Python and Java: The best of both worlds », dans *Proceedings of the 6th International Python Conference*, p. 11–20, San Jose, CA, USA, octobre 1997.

[IEE93] IEEE, *9945-2: 1993 (ISO/IEC) [IEEE/ANSI Std 1003.2-1992 and IEEE/ANSI 1003.2a-1992] Information Technology - Portable Operating System Interface (POSIX®) — Part 2: Shell and Utilities*. IEEE, New York, NY, USA, 1993.

[IKM+97] Ingalls (D.), Kaehler (T.), Maloney (J.) *et al.*, « Back to the future: the story of Squeak, a practical Smalltalk written in itself », dans *OOPSLA '97: Proceedings of the 12th conference on Object-Oriented Programming, Systems, Languages, and Applications*, p. 318–326, Atlanta, Georgia, United States, 1997. ACM Press.

[JL91] Jones (S. L. P.) et Launchbury (J.), « Unboxed values as first class citizens in a non-strict functional language », dans *Proceedings of the 5th ACM conference on Functional Programming Languages and Computer Architecture*, p. 636–666, Cambridge, Massachusetts, United States, 1991. Springer-Verlag New York, Inc.

[Joh79] Johnson (S.), « Lint, a c program checker », dans *UNIX Programmer's Manual 7th Edition, Vol. 2C*. Bell Laboratories, Murray Hill, NJ, 1979.

[Kat79] Katseff (H. P.), « Sdb: a symbolic debugger », dans *UNIX Programmer's Manual 7th Edition, Vol. 2C*. Bell Laboratories, Murray Hill, NJ, 1979.

[KCE98] Kelsey (R.), Clinger (W.) et (Editors) (J. R.), « Revised^5 Report on the algorithmic language Scheme », *ACM SIGPLAN Notices*, vol. 33, n° 9, 1998, p. 26–76.

[Kel95] Kellomaki (P.). « Psd — a portable Scheme debugger », 1995.

[KHH+01] Kiczales (G.), Hilsdale (E.), Hugunin (J.) *et al.*, « An overview of aspectj. », dans *ECOOP '94: proceedings of the 15th European Conference on Object Oriented Programming*, p. 327–353, Budapest, Hungary, juin 2001.

[Kle56] Kleene (S. C.), « Representation of events in nerve nets and finite automata », dans Shannon (C. E.) et McCarthy (J.), éditeurs, *Automata studies*, vol. 34, p. 3–40. Princeton University Press, Princeton, NJ, USA, 1956.

[KR94] Kelsey (R. A.) et Rees (J. A.), « A tractable Scheme implementation », *Lisp and Symbolic Computation*, vol. 7, n° 4, 1994, p. 315–335.

[Laf04] Laforge (G.), « JSR 241: The Groovy scripting language. ». Rapport technique, Oracle, http://www.jcp.org/en/jsr/detail?id=241, mars 2004.

[LaL94] LaLiberte (D.). « Edebug, Emacs LISP debugger », 1994.

[Lar98] Larsen (K. S.), « Regular expressions with nested levels of back referencing form a hierarchy », *Information Processing Letters*, vol. 65, n° 4, 1998, p. 169–172.

[LBR99] Leavens (G. T.), Baker (A. L.) et Ruby (C.), « JML: A notation for detailed design », dans Kilov (H.), Rumpe (B.) et Simmonds (I.), éditeurs, *Behavioral Specifications of Businesses and Systems*, p. 175–188. Kluwer Academic Publishers, 1999.

[Lev04] Levon (J.). « Oprofile - a system profiler for linux ». http://oprofile.sourceforge.net, 2004.

[Lew03] Lewis (B.), « Debugging backwards in time », dans *Proceedings of the Fifth International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, p. 247–258, Ghent, Belgium, septembre 2003. Computer Research Repository.

[LF95] Lieberman (H.) et Fry (C.), « Bridging the gulf between code and behavior in programming », dans *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, p. 480–486, Denver, Colorado, United States, 1995. ACM Press/Addison-Wesley Publishing Co.

[Lid02] Lidin (S.), *Inside Microsoft .NET IL Assembler*. Microsoft Press, Redmond, WA, USA, 2002.

[Lie84] Lieberman (H.), « Steps toward better debugging tools for LISP », dans *LFP '84: Proceedings of the ACM Symposium on LISP and Functional Programming*, p. 247–255, Austin, Texas, United States, 1984. ACM Press.

[LY97] Lindholm (T.) et Yellin (F.), *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, 1997.

[MB77] Maranzano (J. F.) et Bourne (S. R.), « A tutorial introduction to ADB ». Rapport technique, Bell Laboratories, Murray Hill, New Jersey, 1977.

[MB05] Matos (A. A.) et Boudol (G.), « On declassification and the non-disclosure policy. », dans *18th IEEE Computer Security Foundations Workshop*, p. 226–240, juin 2005.

[McC60] McCarthy (J.), « Recursive functions of symbolic expressions and their computation by machine, part i », *Communications of the ACM*, vol. 3, n° 4, 1960, p. 184–195.

[MCKR04] Marceau (G.), Cooper (G. H.), Krishnamurthi (S.) et Reiss (S. P.), « A dataflow language for scriptable debugging », dans *ASE '04:IEEE International Symposium on Automated Software Engineering*, Linz, Austria, septembre 2004. Austrian Computer Society.

[Mey92a] Meyer (B.), « Applying "design by contract". », *IEEE Computer*, vol. 25, n° 10, 1992, p. 40–51.

[Mey92b] Meyer (B.), *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[MKM93] Menapace (J.), Kingdon (J.) et MacKenzie (D.), « The "stabs" debug format ». Rapport technique, Free Software Fundation, Inc., 1993. Contributed by Cygnus Support.

[NBF96] Nichols (B.), Buttlar (D.) et Farrell (J. P.), *PThreads Programming*, coll. « A Nutshell Handbook ». O'Reilly & Associates, Inc., 1996.

[Nie05] Niemeyer (P.), « JSR 241: The BeanShell scripting language. ». Rapport technique, Sun Microsystems Inc., http://www.jcp.org/en/jsr/detail?id=274, juin 2005.

[NPC05] Narayanasamy (S.), Pokam (G.) et Calder (B.), « Bugnet: Continuously recording program execution for deterministic replay debugging », dans *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, p. 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[NS03] Nethercote (N.) et Seward (J.), « Valgrind: A program supervisor rameWork », *Electronic Notes in Theorical Computer Science*, vol. 89, n° 2, 2003, p. 23.

[OCH91] Olsson (R. A.), Crawford (R. H.) et Ho (W. W.), « A dataflow approach to event-based debugging. », *Software — Practice and Experience*, vol. 21, n° 2, 1991, p. 209–230.

[O'H04] O'Hair (K.). « HPROF: A heap/CPU profiling tool in J2SE 5.0 ». http://java.sun.com/developer/technicalArticles/Programming/HPROF.html, novembre 2004.

[Pax90] Paxson (V.). « A survey of support for implementing debuggers ». CS 262, EECS Department, University of California, Berkeley, octobre 1990.

[PCEH05] Prasad (V.), Cohen (W.), Eigler (F. C.) et Hunt (M.), « Locating system problems using dynamic instrumentation », dans *Linux Symposium*, p. 49–64, Ottawa, Canada, juillet 2005.

[PJAB+03] Peyton Jones (S.), Augustsson (L.), Boutel (B.) *et al.*, *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[PL99] Pessaux (F.) et Leroy (X.), « Type-based analysis of uncaught exceptions », dans *POPL'99: Symposium on Principles Of Programming Languages*, p. 276–290, San Antonio, Texas, USA, janvier 1999. ACM Press.

[PLVW98] Pauw (W. D.), Lorenz (D. H.), Vlissides (J. M.) et Wegman (M. N.), « Execution patterns in object-oriented visualization. », dans *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, p. 219–236, Santa Fe, New Mexico, USA, 1998. USENIX Association.

[PS99] Pauw (W. D.) et Sevitsky (G.), « Visualizing reference patterns for solving memory leaks in Java », dans *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, p. 116–134, Lisbon, Portugal, juin 1999. Springer-Verlag.

[Rep91] Reppy (J. H.), « Cml: A higher concurrent language », dans *PLDI '91: Proceedings of the conference on Programming Language Design and Implementation*, p. 293–305, Toronto, Ontario, Canada, juin 1991. ACM Press.

[Ros00] Rossum (G. V.), *Python Reference Manual: Release 1.5.2*. iUniverse, Incorporated, février 2000.

[SB00] Serrano (M.) et Boehm (H.-J.), « Understanding memory aladdress of Scheme programs », *ACM SIGPLAN Notices*, vol. 35, n° 9, 2000, p. 245–256.

[SBS04] Serrano (M.), Boussinot (F.) et Serpette (B.), « Scheme Fair Threads », dans *PPDP '04: Proceedings of the 6th ACM SIGPLAN international confe-*

rence on Principles and Practice of Declarative Programming, p. 203–214, Verona, Italy, janvier 2004. ACM Press.

[SDPK01] Sevitsky (G.), De Pauw (W.) et Konuru (R.), « An information exploration tool for performance analysis of java programs », dans *TOOLS '01: Proceedings of the 38th Technology of Object-Oriented Languages and Systems*, p. 85, Zürich, Switzerland, mars 2001. IEEE Computer Society.

[Ser00] Serrano (M.), « Bee: an integrated development environment for the Scheme programming language », *Journal of Functional Programming*, vol. 10, n° 4, 2000, p. 353–395.

[SF04] Squillace (M.) et Feigenbaum (B.). « Take a shine to JRuby ». http://www-128.ibm.com/developerworks/java/library/j-alj09084/, septembre 2004.

[Sho79] Shoch (J. F.), « An overview of the programming language Smalltalk-72 », *ACM SIGPLAN Notices*, vol. 14, n° 9, 1979, p. 64–73.

[SK05] Strein (D.) et Kratz (H.), « Design and implementation of a high-level multi-language .NET debugger », dans *Proceedings of the 3rd International Conference on .NET Technologies)*, Madrid, Spain, mai 2005.

[SO01] Schinz (M.) et Odersky (M.), « Tail call elimination on the Java Virtual Machine », dans *Proceedings of the BABEL'01 Workshop on Multi-Language Infrastructure and Interoperability.*, vol. 59 (coll. *Electronic Notes in Theoretical Computer Science*), p. 155–168, Firenze, Italy, septembre 2001. Elsevier. `http://www.elsevier.nl/locate/entcs/volume59.html`.

[SOT+00] Suganuma (T.), Ogasawara (T.), Takeuchi (M.) *et al.*, « Overview of the IBM Java Just-in-Time Compiler », *IBM System Journal*, vol. 39, n° 1, 2000, p. 175–193.

[SP91] Stallman (R. M.) et Pesch (R. H.), « Using GDB: A guide to the GNU source-level debugger, GDB version 4.0 ». Rapport technique, Free Software Foundation, Cambridge, MA, 1991.

[SRL90] Sha (L.), Rajkumar (R.) et Lehoczky (J. P.), « Priority inheritance protocols: An approach to real-time synchronization », *IEEE Transactions on Computers*, vol. 39, 1990, p. 1175–1185.

[SS02] Serrano (M.) et Serpette (B.), « Compiling Scheme to JVM bytecode: a performance study », dans *ICFP'02: proceedings of the International Conference on Functional Programming*, p. 259–270, Pittsburgh, PA, USA, octobre 2002. ACM Press.

[Sta81] Stallman (R. M.), « EMACS: the extensible, customizable self-documenting display editor », *ACM SIGPLAN Notices*, vol. 16, n° 6, 1981, p. 147–156.

[Sun02] Sun Microsystem Inc., éditeur, *Solaris Modular Debugger*. Iuniverse Inc, mai 2002.

[SW95] Serrano (M.) et Weis (P.), « Bigloo: A portable and optimizing compiler for strict functional languages », dans *SAS '95: proceedings of the international Static Analysis Symposium*, vol. 983 (coll. *Lecture Notes in Computer Science*), p. 366–381, Glasgow, UK, septembre 1995. Springer-Verlag.

[TA95] Tolmach (A. P.) et Appel (A. W.), « A debugger for standard ML », *Journal of Functional Programming*, vol. 5, n° 2, 1995, p. 155–200.

[TD86] Tuthill (B.) et Dunlap (K.), « Debugging with dbx », dans *UNIX Programmer's Supplementary Documents, Vol. 1*, avril 1986.

[TH02] Trentelman (K.) et Huisman (M.), « Extending jml specifications with temporal logic », dans *AMAST '02: proceedings of the conference on Algebraic Methodology And Software Technology*, vol. 2422 (coll. *Lecture Notes in Computer Science*), Berlin, Germany, 2002. Springer-Verlag.

[THL02] Tikir (M. M.), Hollingsworth (J. K.) et Lueh (G.-Y.), « Recompilation for debugging support in a jit-compiler », dans *PASTE '02: Proceedings of the workshop on Program Analysis for Software Tools and Engineering*, p. 10–17, Charleston, South Carolina, USA, 2002. ACM Press.

[US87] Ungar (D.) et Smith (R. B.), « Self: The power of simplicity », dans *OOPSLA '87: Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*, p. 227–242, Orlando, Florida, USA, octobre 1987. ACM Press.

[War04] Warkus (M.), *The Official GNOME 2 Developer's Guide*. No Starch Press, Reading, MA, USA, 2004.

[Wei82] Weiser (M.), « Programmers use slices when debugging », *Communications of the ACM*, vol. 25, n° 7, 1982, p. 446–452.

[WTE05] Williams (A.), Thies (W.) et Ernst (M. D.), « Static deadlock detection for Java libraries », dans *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, p. 602–629, Glasgow, Scotland, juillet 2005.

[Zel78] Zelkowitz (M. V.), « Perspectives on software engineering », *ACM Computing Surveys*, vol. 10, n° 2, 1978, p. 197–216.

[Zel83] Zellweger (P. T.), « An interactive high-level debugger for control-flow optimized programs », dans *SIGSOFT '83: Proceedings of the symposium on High-level debugging*, p. 159–172, Pacific Grove, California, mars 1983. ACM Press.

[Zel84] Zellweger (P. T.), *Interactive Source-Level Debugging of Optimized Programs*. Thèse de doctorat, University of California, Xerox Palo Alto Research Center, mai 1984.

[ZL77] Ziv (J.) et Lempel (A.), « A universal algorithm for sequential data compression », *IEEE Transactions on Information Theory*, vol. 23, n° 3, 1977, p. 337–343.

[ZL96] Zeller (A.) et Lutkehaus (D.), « DDD - a free graphical front-end for UNIX debuggers », *ACM SIGPLAN Notices*, vol. 31, n° 1, 1996, p. 22–27.

[ZS95] Zhao (Q. A.) et Stasko (J. T.), « Visualizing the execution of threads-based parallel programs ». Rapport technique n° GIT-GVU-95-01, College of Computing, George Institute of Technology, 1995.

« *Toute existence connaît son jour de traumatisme primal, qui divise cette vie en un avant et un après et dont le souvenir même furtif suffit à figer dans une terreur irrationnelle, animale et inguérissable.* »

A. Nothomb, *Stupeur et tremblements*

Résumé

Cette thèse est consacrée à l'amélioration des débogueurs symboliques pour tenir compte des spécificités des langages de haut niveau, notamment leur compilation délicate vers des plates-formes d'exécution généralistes. Ces travaux ont conduit à la réalisation de Bugloo, un débogueur multi-langages pour la machine virtuelle Java. Les implanteurs de langages peuvent facilement étendre ce débogueur en fonction des spécificités de leur compilation, comme l'encodage des noms d'identificateurs ou l'utilisation d'un système de type *ad-hoc*.

Deux nouveaux mécanismes sont proposés pour masquer les artefacts introduits dans la pile par compilation des langages de haut niveau. Le premier est un algorithme qui utilise des règles fournies par les implanteurs de langage pour identifier des blocs d'activations indésirables dans la pile et pour reconstruire une vue logique dans laquelle ces détails de compilation ont été expurgés. Cette technique permet en outre de visualiser correctement des piles contenant à la fois du code natif et du code interprété. Le second mécanisme sert à contrôler l'exécution pas-à-pas, afin de ne jamais s'arrêter dans les fonctions intermédiaires engendrées par le compilateur.

Divers outils de débogage mémoire sont présentés, dont un profileur d'allocation adapté aux langages de haut niveau et à leur compilation délicate. Cet outil emploie les techniques de représentation logique de pile afin de produire des statistiques contenant uniquement des fonctions utilisateur, tout en recensant correctement les allocations effectuées par les fonctions intermédiaires engendrées par la compilation.

Durant ces travaux, un support de débogage complet a été développé pour le langage Bigloo, un dialecte du langage fonctionnel Scheme. Des expérimentations similaires ont étés menées sur les langages ECMAScript et Python. Les résultats obtenus montrent que les techniques de représentations virtuelles développées s'appliquent efficacement quel que soit le schéma de compilation adopté, y compris lorsque les programmes sont composés de plusieurs langages de haut niveau.

Mots clés : débogage symbolique, langages de haut niveau, représentations virtuelles, JVMTI, Scheme, machine virtuelle Java.

Abstract

This thesis is devoted to improving symbolic debuggers so that they can deal with the specifics of high-level languages, in particular their complex compilation to general-purpose execution platforms. This has led to the implementation of Bugloo, a language-agnostic debugger for the Java Virtual Machine. Language implementors can extend this debugger to add support for their compilation scheme, for example special name mangling or custom type system.

Two novels mechanisms are introduced in order to mask artifacts that show up in the stack due to the compilation of high-level languages. The first one is an algorithm that uses special rules designed by language implementors to locate unwanted frames in the stack and to reconstruct a logical view of this stack where the compilation details have been filtered out. With this technique, stacks that contain both native and interpreted code can be visualized seamlessly. The second mechanism provides a means to control single stepping, in order to disallow execution to stop in intermediate functions that were generated by the compiler.

Various memory debugging tools are presented, including a memory profiler designed to support high-level languages and their complex compilation. This tool uses logical stack representation techniques in order to generate statistics that only contain user-level functions. Moreover, these statistics correctly incorporate the allocations that occurred within intermediate functions generated by the compiler.

During this work, a complete debugging support has been developed for Bigloo, a dialect of the functional language Scheme. Other experiments have been conducted on the high-level languages ECMAScript and Python. Results show that our virtualization techniques can be efficiently applied whatever the compilation scheme is. Moreover, they remain effective when programs are composed of several high-level languages.

Keywords: source-level debuggers, high-level languages, virtualization, JVMTI, Scheme, Java Virtual Machine.